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.

Note that the following URLs in this document are not available:
http://www.necel.com/
http://www2.renesas.com/

Please refer to the following instead:
Development Tools | http://www.renesas.com/tools
Download | http://www.renesas.com/tool_download

For any inquiries or feedback, please contact your region.
http://www.renesas.com/inquiry

RENESAS

Notice

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

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

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

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

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

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

RENESAS

User's Manual

CC78K0R Ver. 2.00

C Compiler

Language

Target Device
78K0R Microcontrollers

Document No. U18548EJ1V0UM00 (1st edition)
Date Published October 2007

Printed in Japan

[MEMO]

[MEMO]

- **The information in this document is current as of October, 2007. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.
- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".
  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

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

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

M8E 02. 11-1

[MEMO]

INTRODUCTION

The **CC78K0R C Compiler** (hereinafter referred to as this C compiler) was developed based on **CHAPTER 2 ENVIRONMENT** and **CHAPTER 3 LANGUAGE** in the **Draft Proposed American National Standard for Information Systems — Programming Language C** (December 7, 1988). Therefore, by compiling C source programs conforming to the ANSI standard with this C compiler, applied products for the 78K0R Microcontroller can be developed.

The **CC78K0R C Compiler Language** (this manual) has been prepared to give those who develop software by using this C compiler a correct understanding of the basic functions and language specifications of this C compiler.

This manual does not cover how to operate this C compiler. Therefore, after you have comprehended the contents of this manual, read the **CC78K0R Ver. 2.20 C Compiler Operation (U18549E)**.

For the architecture of 78K0R Microcontroller, refer to the user's manual of each product of 78K0R Microcontroller.

[Target Devices]

Software for the 78K0R Microcontroller microcontrollers can be developed with this C compiler.
Note that an optional device file corresponding to a target device is necessary.

[Readers]

Although this manual is intended for those who have read the user's manual of the microcontroller subject to software development and have experience in software programming, the readers need not necessarily have a knowledge of C compilers or C language. Discussions in this manual assume that the readers are familiar with software terminology.

[Organization]

This manual consists of the following 13 chapters and appendixes:

Chapter 1 - GENERAL
Outlines the general functions of C compilers and the performance characteristics and features of this C compiler.

Chapter 2 - CONSTRUCTS OF C LANGUAGE
Explains the constituting elements of a C source module file.

Chapter 3 - DECLARATION OF TYPES AND STORAGE CLASSES
Explains the data types and storage classes used in C and how to declare the type and storage class of a data object or function.

Chapter 4 - TYPE CONVERSIONS
Explains the conversions of data types to be automatically carried out by this C compiler.

Chapter 5 - OPERATORS AND EXPRESSIONS
Describes the operators and expressions that can be used in C and the precedence of operators.

Chapter 6 - CONTROL STRUCTURES OF C LANGUAGE
Explains the program control structures of C and the statements to be executed in C.

Chapter 7 - STRUCTURES AND UNIONS
Explains the concept of structures and unions and how to refer to structure and union members.

Chapter 8 - EXTERNAL DEFINITIONS
Describes the types of external definitions and how to use external declarations.

Chapter 9 - PREPROCESSOR DIRECTIVES (COMPILER DIRECTIVES)
Details the types of preprocessing directives and how to use each preprocessor directive.

Chapter 10 - LIBRARY FUNCTIONS
Details the types of C library functions and how to use each library function.

Chapter 11 - EXTENDED FUNCTIONS
Explains the extended functions of this C compiler to make the most of the target device.

Chapter 12 - REFERENCING THE ASSEMBLER
Describes the method of linking a C source program with a program written in Assembly language.

Chapter 13 - EFFECTIVE UTILIZATION OF COMPILER
Outlines how to effectively use this C compiler.

APPENDIXES
Contains a list of labels for saddr area, a list of segment names, a list of runtime libraries, a list of library stack consumption, a list of maximum interrupt disabled time for libraries, and index for quick reference.

[How to Read This Manual]

- For those who are not familiar with C compilers or C language:
  Read from **Chapter 1**, as this manual covers from the program control structures of C to the extended functions of this C compiler. In **Chapter 1**, an example of C source program is used to show the reference part in this manual.
- For those who are familiar with C compilers or C language:
  The language specifications of this C compiler conform to the **ANSI Standard C**. Therefore, you may start from **Chapter 11** that explains the extended functions unique to this C compiler. When reading **Chapter 11**, also refer to the user's manual supplied with the target device in the 78K0R Microcontroller if necessary.

[Related Documents]

The table below shows the documents (such as user's manuals) related to this manual. The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such.

Documents related to development tools (user's manuals)

Document Name		Document No.
CC78K0R Ver. 2.00 C Compiler	Operation	U18549E
	Language	This document
RA78K0R Ver. 1.20 Assembler Package	Operation	U18547E
	Language	U18546E
SM+ System Simulator	Operation	U18010E
PM+ Ver. 6.30 Project Manager		U18416E
ID78K0R-QB Ver. 3.20 Integrated Debugger	Operation	U17839E

[Reference]

Draft Proposed American National Standard for Information Systems - Programming Language C (December 7, 1988)

[Terms]

RTOS = **78K0R Microcontroller Real-time OS RX78K0R**

[Conventions]

The following symbols and abbreviations are used in this manual:

Symbol	**Meaning**
⋮	: Continuation (repetition) of data in the same format
" "	: Characters enclosed in a pair of double quotes must be input as is.
' '	: Characters enclosed in a pair of single quotes must be input as is.
:	: This part of the program description is omitted.
/	: Delimiter
\	: Backslash
[ ]	: Parameters in square brackets may be omitted.

CONTENTS

LIST OF FIGURES

Figure No. Title , Page

LIST OF TABLES

Table No. Title , Page

CHAPTER 1 GENERAL

This chapter explains the roles of the CC78K0R at the time of system development and functional outlines of the C Compiler.

The 78K0R C Compiler is a language processing program which converts a source program written in the C language for the 78K0R or ANSI-C into machine language. By the 78K0R C compiler, object files or assembler source files for the 78K0R can be obtained.

1.1 C Language and Assembly Language

To have a microcontroller do its job programs and data are necessary. These programs and data must be written by a user being and stored in the memory section of the microcontroller. Programs and data that can be handled by the microcontroller are nothing but a set or combination of binary numbers that is called machine language.

An assembly language is a symbolic language characterized by one-to-one correspondence of its symbolic (mnemonic) statements with machine language instructions. Because of this one-to-one correspondence, the assembly language can provide the computer with detailed instructions (for example, to improve I/O processing speed). However, this means that the user must instruct each and every operation of the computer. For this reason, it is difficult for him or her to understand the logic structure of the program at glance and the user is likely to make errors in coding.

High-level languages were developed as substitutes for such assembly languages. The high-level languages include a language called C that allows the user to write a program without regard to the architecture of the computer.

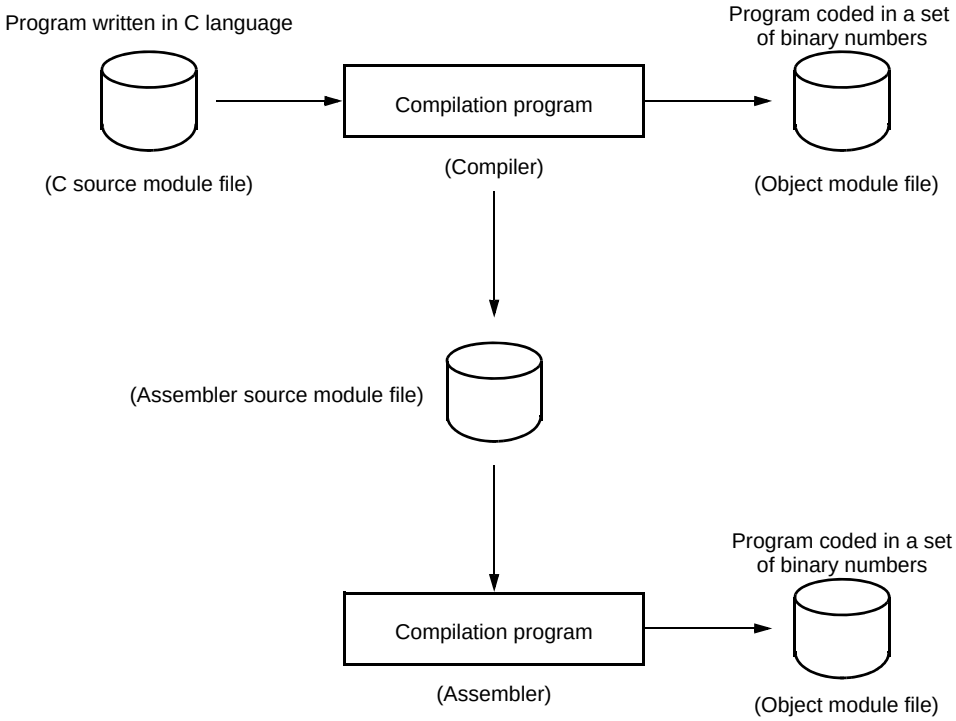
As compared with assembly language programs, it can be said that programs written in C have easy-to-understand logic structure.

C has a rich set of parts called functions for use in creating programs. In other words, the user can write a program by combining these functions.

C is characterized by its ease of understanding by user beings. However, understanding of languages by the microcontroller cannot be extended up to a program written in C. Therefore, to have the computer understand the C language program, another program is required to translate C language statements to the corresponding machine language instructions. A program that translates the C language into machine language is called a C compiler.

C compiler accepts C source modules as inputs and generates object modules or assembler source modules as outputs. Therefore, the user can write a program in C and if he or she wishes to instruct the computer up to details of program execution, the C source program can be modified in assembly language. The flow of translation by C compiler is illustrated below.

Figure 1-1 Flow of Compilation



1.2 Program Development Procedure by C Compiler

Product (program) development by the C compiler requires a linker which unites together object module files created by the compiler, a librarian which creates library files, and a debugger which locates and corrects bugs (errors or mistakes) in each created C source program.

1.2.1 Software required

software requiredThe software required in connection with C compiler is shown below.

- Editor

  for source module file creation

- RA78K0R assembler package

Program Nmae	Function
Assembler	For converting assembly language into machine language
Linker	For linking object module files For determining location address of relocatable segment
Object converter	For conversion to HEX-format object module file
Librarian	For creating library files
List converter	For output of an absolute assemble list file
PM+	Integrated development environment platform

- Integrated debugger (for 78K0R)

  for debugging C source module files

1.2.2 Product development procedure

The product development procedure by the C compiler is as shown below.

(1) Divide the product into functions.

(2) Creates a C source module for each function.

(3) Translates each C source module.

(4) Registers the modules to be used frequently in the library.

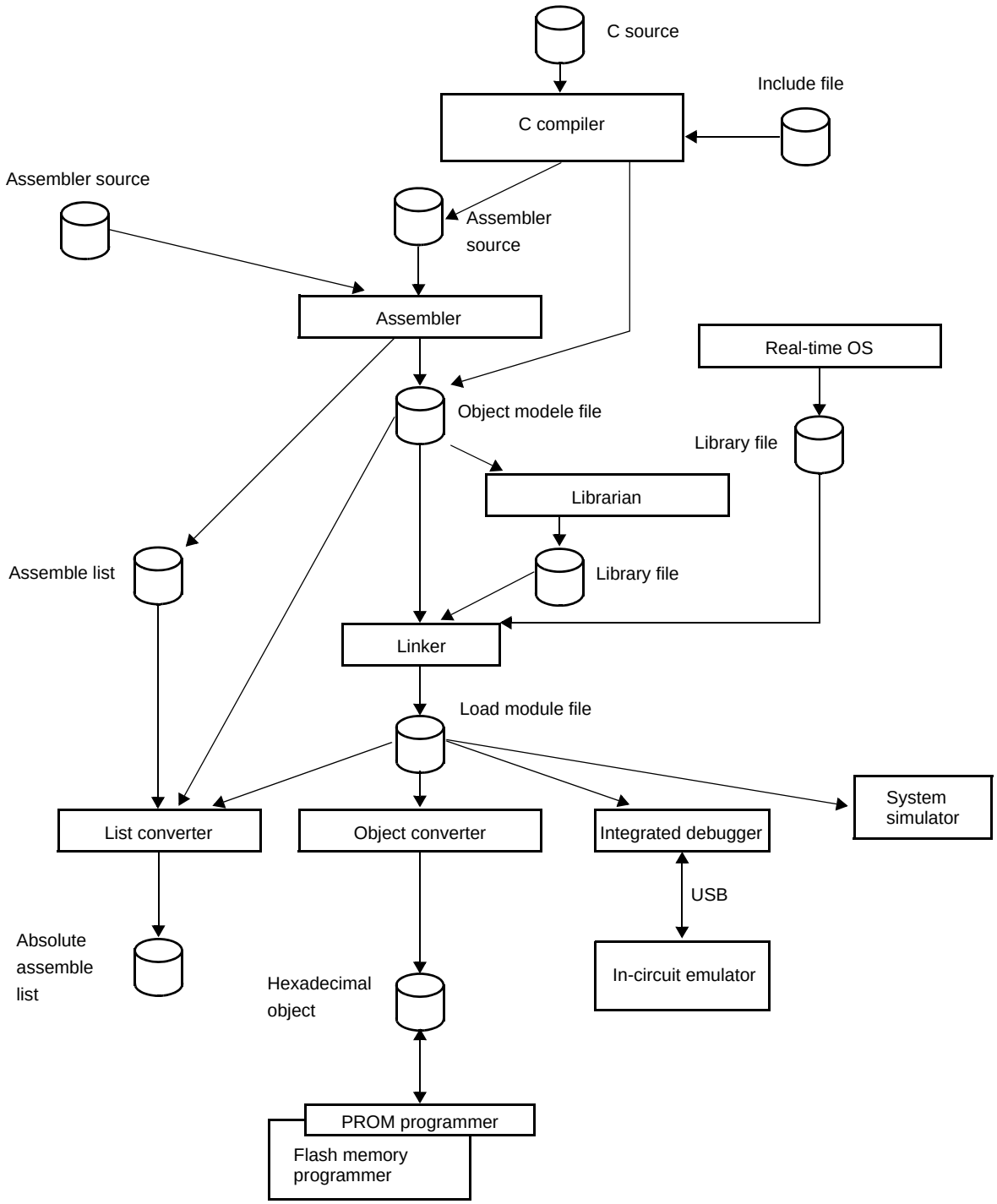
(5) Links object module files.

(6) Debugs each module.

(7) Converts object modules into HEX-format object files.

As mentioned earlier, C compiler translates (compiles) a C source module file and creates an object module file or assembler source module file. By manually optimizing the created assembler source module file and embedding it into the C source, efficient object modules can be created. This is useful when high-speed processing is a must or when modules must be made compact.

Figure 1-2 Program Development Procedure by CC78K0R

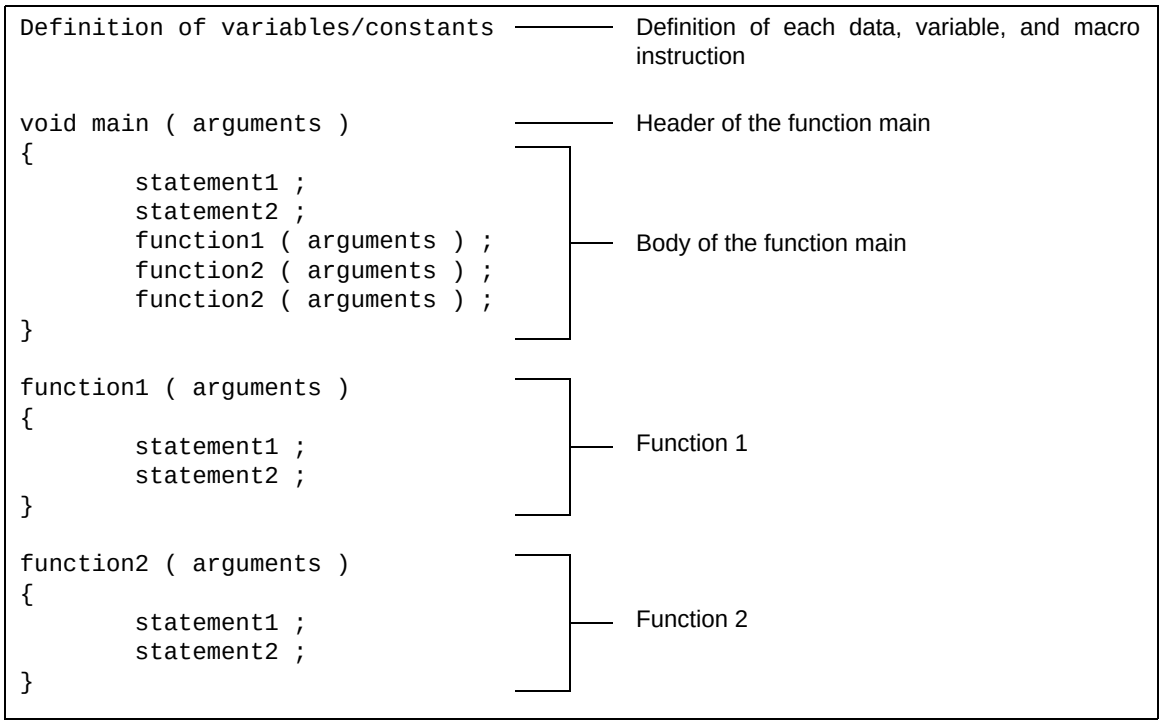


1.3 Basic Structure of C Source Program

1.3.1 Program format

A C language program is a collection of functions. These functions must be created so that they have independent special-purpose or characteristic actions. All C language programs must have a function main which becomes the main routine in C and is the first function that is 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 xx xx : Preprocessor directive (macro definition) */
#define FALSE   0   /* #define xx xx : Preprocessor directive (macro definition) */
#define SIZE    200 /* #define xx xx : Preprocessor directive (macro definition) */

void    printf ( char* , int ) ;  /* xx xx ( xx , xx ) : Function prototype declarator */
void    putchar ( char ) ;        /* xx xx ( xx ) :     Function prototype declarator */

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

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

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

    for ( i = 0 ; i <= SIZE ; i ++ ) {
        if ( mark [ i ] ) {
            prime = i + i + 3 ;           /* xx = xx + xx + xx :    Operator */
            printf ( "%6d" , prime ) ;    /* xx ( xx ) ; :          Operator */

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

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

    j = i ;
    ss = s ;
}

void    putchar ( char c ) {
    char    d ;

    d = c ;
}
```

(1) Declaration of type and storage class

The data type and storage class of an identifier that indicates a data object are declared.

For details, see "CHAPTER 3 DECLARATION OF TYPES AND STORAGE CLASSES".

(2) Operator and expression

These are the statements, which instructs the compiler to perform an arithmetic operation, logical operation, assignment, or like.

For details, see "CHAPTER 5 OPERATORS AND EXPRESSIONS".

(3) Control structure

This is a statement that specifies the program flow. C has several instructions for each of control structures such as Conditional control, Iteration, and Branch.

For details, see "CHAPTER 6 CONTROL STRUCTURES OF C LANGUAGE".

(4) Structure or union

A structure or union is declared. A structure is a data object that contains several subobjects or members that may have different types. A union is defined when two or more variables share the same memory.

For details, see "CHAPTER 7 STRUCTURES AND UNIONS".

(5) External definition

A function or external object is declared. A function is 1 element when a C language program is divided by a special-purpose or characteristic action. A C program is a collection of these functions.

For details, see "CHAPTER 8 EXTERNAL DEFINITIONS".

(6) Preprocessor directive

This is an instruction for the compiler. #define instructs the compiler to replace a parameter which is the same as the first operand with the second operand if the parameter appears in the program.

For details, see "CHAPTER 9 PREPROCESSOR DIRECTIVES (COMPILER DIRECTIVES)".

(7) Declaration of function prototype

The return value and argument type of a function are declared.

1.4 Quantitative Limits for C Compiler

Before you set your hand to the development of a program, keep in mind the points (limit values or minimum guaranteed values) summarized in Table 1-1 below.

Table 1-1 Quantitative Limits for C Compiler

Item	Quantitative Limits
Nesting level of compound statements, looping statements, or conditional control statements	45 levels
Nesting of conditional translations	255 levels
Number of arithmetic type, structure type, pointer to qualify union type or incomplete type, array, and function declarator in a declaration (or any combination of these)	12 levels
Nesting of parentheses per expression	32 levels
Number of characters which have a meaning as a macro name	256 characters
Number of characters which have a meaning as an internal or external symbol name	249 characters
Number of symbols per translation unit	1,024 symbols[Note 1]
Number of symbols which has block scope within a block	255 symbols
Number of macros per translation unit	32,767macros
Number of parameters per function definition or function call	39 parameters[Note 1]
Number of parameters per macro definition or macro call	31 parameters
Number of characters per logical source line	2,048 characters[Note 1]
Number of characters within a string literal after linkage	509 characters[Note 1]
Size of 1 data object	65,535 bytes
Nesting of #include directives	50 levels
Number of case labels per switch statement	257 labels
Number of lines per file	Approx. 65,535 lines[Note 1]
Nest of function calls	40 levels[Note 1]
Number of labels within a function	33 labels
Total size of code, data, and stack segments per object module	Varies depending on the memory model[Note 2]
Number of members per structure or union	256 members
Number of enum constants per enumeration	255 constants
Nest of structures or unions inside a structure or union	15 levels
Nest of initializer elements	15 levels
Number of function definitions per translation unit	4,095

Table 1-1 Quantitative Limits for C Compiler

Item	Quantitative Limits
Level of the nest of declarator enclosed with parentheses inside a complete declarator.	591 Note 1
Nest of macros	200
Number of -i include file path specifications	64

Note 1 This is a guaranteed value. Values larger than this value may be specifiable, but the operation is not guaranteed.

Note 2 The maximum value varies as follows, depending on the memory model selected.

Memory Model	Maximum Value
Small model	Code portion: 64 KB, data portion: 64 KB; 128 KB in total
Medium model	Code portion: 1 MB, data portion: 64 KB
Large model	Code portion: 1 MB, data portion: 1 MB

1.5 Features of C Compiler

The CC78K0R has extended functions for CPU code generations that are not supported by the ANSI (American National Standards Institute) Standard C. The extended functions of the C compiler allow the special function registers for the 78K0R to be described at the C language level and thus help shorten object code and improve program execution speed.

Outlined here are the following extended functions to help shorten object code and improve execution speed:

Table 1-2 Methods to Improve Execution Speed

Method	Extended Functions
Functions can be called using the callt table area.	callt /__callt functions
Variables can be allocated to registers.	Register variables
Variables can be allocated to the saddr area.	sreg/__sreg
sfr names can be used.	sfr area
Functions that do not output code for stack frame formation can be created.	norec/__leaf functions
An assembly language program can be described in a C source program	ASM statements
Accessing the saddr or sfr area can be made on a bit-by-bit basis.	bit type variables, boolean/__boolean type variables
A bit field can be specified with unsigned char type.	Bit field declaration
The code to multiply can be directly output with inline expansion.	Multiplication function
Codes that achieve faster execution as well as smaller size and being compatible with the CC78K0, can be generated.	Division function
The code to rotate can be directly output with inline expansion.	Rotate function
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 manipulation function

An outline of the expansion functions of the CC78K0R is shown below.

For details of each expansion function, please refer to "CHAPTER 11 EXTENDED FUNCTIONS".

Table 1-3 List of Extended Functions

Extended Functions	Outline
callt functions (callt/__callt)	Functions can be called by using the callt table area. The address of each function to be called (this function is called a callt function) is stored in the callt table from which it can be called later. This makes code shorter than the ordinary call instruction and helps shorten object code.

Table 1-3 List of Extended Functions

Extended Functions	Outline
Register variables (register)	Variables declared with the register storage class specifier are allocated to the register or saddr area. Instructions to the variables allocated to the register or saddr area are shorter in code length than those to memory. This helps shorten object and improves program execution speed as well.
How to use the saddr area (sreg/ __sreg)	Variables declared with the keyword sreg can be allocated to the saddr area. Instructions to these sreg variables are shorter in code length than those to memory. This helps shorten object code and also improves program execution speed. Variables can be allocated to the saddr area also by option.
How to use the sfr area (sfr)	By declaring use of sfr names, manipulations on the sfr area can be described at the C source file.
bit type variables, boolean type variables (bit/boolean/ __boolean)	Variables having a 1-bit storage area are generated. By using the bit type variable or boolean/__boolean type variable, the saddr area can be accessed in bit units. The boolean/__boolean type variable is the same as the bit type variable in terms of both function and usage.
ASM statements (#asm - #endasm/__asm)	The assembler source program described by the user can be embedded in an assembler source file to be output by the CC78K0R.
Kanji (2-byte character) (/* kanji */, // kanji)	Kanji code (2-byte characters) can be described in comments in C source files. Shift JIS or EUC can be selected as the character code. Prohibiting the use of kanji codes can also be selected.
Interrupt functions (#pragma vect/#pragma interrupt)	The preprocessor directive outputs a vector table and outputs an object code corresponding to the interrupt. This directive allows programming of interrupt functions in the C source level.
Interrupt function qualifier (__interrupt, __interrupt_brk)	This qualifier allows the setting of a vector table and interrupt function definitions to be described in a separate file.
Interrupt functions (#pragma DI, #pragma EI)	An interrupt disable instruction and an interrupt enable instruction are embedded in the object.
CPU control instruction (#pragma HALT/STOP/BRK/ NOP)	Each of the following instruction is embedded in the object: - halt Instruction - stop Instruction - brk instruction - nop instruction
Bit field declaration	By specifying a bit field to be unsigned char type, the memory can be saved, object code can be shortened, and execution speed can be improved.
Changing compiler output section name (#pragma section ...)	By changing the compiler section output name, the section can be independently allocated with a linker.
Binary constant (Binary constant 0bxxx)	Binary can be described in the C source.
Module name changing function (#pragma name)	Object module names can be freely changed in the C source.

Table 1-3 List of Extended Functions

Extended Functions	Outline
Rotate function (#pragma rot)	The code to rotate the value of an expression to the object can be directly output with inline expansion.
Multiplication function (#pragma mul)	The code to multiply the value of an expression to the object can be directly output with inline expansion. This function can shorten the object code and improve the execution speed.。
Division function (#pragma div)	Codes that are compatible with the CC78K0 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.
Data insertion function (#pragma opc)	Constant data is inserted in the current address. Specific data and instructions can be embedded in the code area without using assembler description.
Interrupt handler for RTOS (#pragma rtos_interrupt ...)	Interrupt handlers for the RX78K0R (real-time OS) can be described.
Interrupt handler qualifier for RTOS (__rtos_interrupt)	This qualifier allows the interrupt handler description and the vector setting for the RX78K0R (real-time OS) made in separate files.
Task function for RTOS (#pragma rtos_task)	Specified functions are interpreted as the tasks for the RX78K0R (real-time OS) by #pragma instruction. This allows the description of task function for real-time OS with better code-efficiency in the C source level.
Flash area allocation method (-zf)	A program can be allocated to the flash area by specifying the -zf option during compilation, or a program can be used in combination with the object created in the boot area without specifying the -zf option.
Flash area branch table (#pragma ext_table)	The startup routine, allocation of interrupt function to the flash area, and function call from the boot area to the flash area are performed by specifying the first address of the flash area branch table by the #pragma directive.
Function of function call from boot area to flash area (#pragma ext_func)	A function can be called from the boot area to the flash area by specifying the function name and the ID value in the flash area called from the boot area by the #pragma directive.
Firmware ROM function (__flash)	During the prototype declaration of the interface library, manipulations regarding the firmware ROM function can be described in C source level by adding the __flash attribute to the first address.
Method of int expansion limitation of argument/return value (-zb)	By specifying the -zb option during compilation, the object code can be shortened and execution speed can be improved.
Memory manipulation function (#pragma inline)	By #pragma inline directive, an object file is generated by the output of the standard library functions memcpy and memset with direct inline expansion instead of function call. This function can improve the execution speed.
Absolute address allocation specification (__directmap)	By declaring __directmap in the module in which the variable to be allocated to an absolute address is to be defined, one or more variables can be allocated to the same arbitrary address.
near/far area specification	By adding the __near or__far type qualifier when declaring a function or variable, the location of the function or variable can be specified explicitly.

Table 1-3 List of Extended Functions

Extended Functions	Outline
Memory model specification	By specifying the -ms, -mm, -mc, or -ml option during compilation, the location of the function or variable can be specified by specifying the memory model.

CHAPTER 2 CONSTRUCTS OF C LANGUAGE

This chapter explains the constituting elements of a C source module file.

A C source module file consists of the following tokens (distinguishable units in a sequence of characters).

Keywords	Identifiers	Constants
String literal	Operators	Delimiters
Header name	No. of preprocesses	Comment

The tokens used in the C program description example are shown below.

```
#include    "expand.h"

extern      void    testb ( void ) ;        /* extern :        Keyword */
extern      void    chgb ( void ) ;
extern      bit     data1 ;                 /* data1, data2 :  Identifiers */
extern      bit     data2 ;

void    main ( void ) {                     /* void :          Keyword */
        data1 = 1 ;                         /* 1 :             Constant */
        data2 = 0 ;                         /* 0 :             Constant */

        while ( data1 ) {                   /* while :         Keyword */
                data1 = data2 ;             /* { } :           Delimiter */
                testb ( ) ;                 /* = :             Operator */
        }
        if ( data1 && data2 ) {             /* if :            Keyword */
                chgb ( ) ;                  /* && :            Operator */
        }                                   /* ( ) :           Operator */
}

void    lprintf ( char *s , int i ) {       /* lprintf :       Identifiers */
                                            /* char, int :     Keyword */
        int     j ;                         /* s, i :          Identifiers */
        char    *ss ;                       /* * :             Operator */
        j = i ;
        ss = s ;
}
            :
```

2.1 Character Sets

2.1.1 Character sets

Character sets to be used in C programs include a source character set to be used to describe a source file and an execution character set to be interpreted in the execution environment.

The value of each character in the execution character set is represented by JIS code.

The following characters can be used in the source character set and execution character set.

All full-size characters and half-size katakana (including half-size punctuation marks) can be described in comments.

Table 2-1 List of Characters that Can Be Used in Character Set

26 uppercase letters	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
26 lowercase letters	a b c d e f g h i j k l m n o p q r s t u v w x y z
10 figures	0 1 2 3 4 5 6 7 8 9
29 graphic characters	! " # % & ' ( ) * + , - . / : ; < = > ? [¥] ^ _ { | } ~
Nonprintable control characters which indicate Space, Horizontal Tab, Vertical Tab, Form Feed, etc.	

2.1.2 Multi-byte character

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.

2.1.3 ESCAPE sequences

Nongraphic characters used for control characters as for alert, formfeed, and such are represented by ESCAPE sequences.

Each ESCAPE sequence consists of the \ sign and an alphabetic character.

Nongraphic characters represented by ESCAPE sequences are shown in the table below.

Table 2-2 List of ESCAPE Sequences

ESCAPE Sequence	Meaning	Character Code
\a	Alert	07H
\b	Backspace	08H
\f	Formfeed	0CH
\n	New Line	0AH
\r	Carriage Return	0DH
\t	Horizontal Tab	09H
\v	Vertical Tab	0BH

2.1.4 Trigraph sequences

When a source file includes a list of the 3 characters (called "trigraph sequence") shown in the left column of the table below, the list of the 3 characters is converted into the corresponding single character shown in the right column.

The trigraph sequence is enabled when compiler option -za (the option that disables the functions which do not comply with ANSI specifications and enables a part of functions of ANSI specifications) is specified.

Table 2-3 List of Trigraph Sequence

Trigraph Sequence	Meaning
??=	#
??(	[
??/	\
??)	]
??’	^
??<	{
??!	|
??>	}
??-	~

2.2 Keywords

2.2.1 ANSI-C keywords

The following tokens are used by the C compiler as keywords and thus cannot be used as labels or variable names.

Table 2-4 List of ANSI-C Keywords

auto	break	case	char	const	continue	default
do	double	else	enum	extern	for	float
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

2.2.2 Keywords added for the CC78K0R

In the CC78K0R the following tokens have been added as keywords to implement its expanded functions.

These tokens cannot be used as labels or variable names nor can ANSI (when an uppercase character is included, the token is not regarded as a keyword).

Keywords which do not start with "__" can be made invalid by specifying the option (-za) that enables only ANSI-C language specification.

callf, __callf, noauto, __banked, __non_banked, __BANK0-15, __mxcall, __pascal, __temp, norec, __leaf are taken as keywords for compatibility with the CC78K0.

Table 2-5 List of Keywords Added for CC78K0R

Keywords	Usage
__callt/callt	Declaration of callt function
__callf/callf	Declaration of callf function
__sreg/sreg	Declaration of sreg variable
noauto	Declaration of noauto function
__leaf/norec	Declaration of leaf/norec function
bit	Declaration of bit type variable
__boolean/boolean	Declaration of boolean type variable
__interrupt	Hardware interrupt function
__interrupt_brk	Software interrupt function
__banked, __non_banked	Bank interface[Note 1]
__BANK0-15	Bank functions at constant addresses
__asm	asm statement
__rtos_interrupt	Interrupt handler for real-time OS
__pascal	Pascal function
__flash	Firmware ROM function
__flashf	__flashf function[Note 2]
__directmap	Absolute address allocation specification
__temp	Temporary variable
__near, __far	Memory allocation area specification
__mxcall	__mxcall function[Note 3]

Note 1 Reserved keyword for function information files. Do not describe this keyword in the C source.

Note 2 Reserved keyword by of the CC78K0R. This keyword must not be used by users.

Note 3 Reserved keyword for interface with MX. This keyword must not be used by users.

2.3 Identifiers

An identifier is the name that you give to a variable such as:

Table 2-6 List of Identifiers

Function
Object
Tag of structure, union, or enumeration type
Member of structure, union, or enumeration type
typedef name
Label name
Macro name
Macro parameter

Each identifier can consist of uppercase letters, lowercase letters, or numeric characters including underscores.

The following characters can be used as identifiers:

There is no restriction for the maximum length of the identifier. In the CC78K0R, however, only the first 249 characters can be identified.

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

All identifiers must begin with other than a numerical character (namely, a letter or an underscore) and must not be the same as any keyword.

2.3.1 Scope of identifiers

The range of an identifier within which its use becomes effective is determined by the location at which the identifier is declared. The scope of identifiers is divided into the following 4 types:

- Function scope
- File scope
- Block scope
- Function prototype scope

```
extern __boolean data1 , data2 ;   /* data1, data2 : File scope */

void   testb ( int x ) ;           /* x :           Function prototype scope */

void   main ( void ) {
       int     cot ;               /* cot :         Block scope */
       data1 = 1 ;
       data2 = 0 ;

       while ( data1 ) {
                data1 = data2 ;
           j1 :                    /* j1 :          Function scope */
                testb ( cot ) ;
       }
}

void    testb ( int x ) {          /* x :           Block scope */
     :
}
```

(1) Function scope

Function scope refers to the entirety within a function.

An identifier with function scope can be referenced from anywhere within a specified function. Identifiers that have function scope are label names only.

(2) File scope

File scope refers to the entirety of a translation (compiling) unit. Identifiers that are declared outside a block or parameter list all have file scope. An identifier that has file scope can be referenced from anywhere within the program.

(3) Block scope

Block scope refers to the range of a block (a sequence of declarations and statements enclosed by a pair of curly braces { } which begins with the opening brace and ends with the closing brace.

Identifiers that are declared inside a block or parameter list all have block scope. An identifier that has block scope is effective until the innermost brace pair including the declaration of the identifier is closed.

(4) Function prototype scope

Function prototype scope refers to the range of a declared function from its beginning to the end. Identifiers that are declared inside a parameter list within a function prototype all have function prototype scope. An identifier that has function prototype scope is effective within a specified function.

2.3.2 Linkage of identifiers

The linkage of an identifier refers to that the same identifier declared more than once in different scopes or in the same scope can be referenced as the same object or function.

An identifier by being linked is regarded to be one and the same.

An identifier may be linked in the following 3 different ways: External linkage, Internal linkage and No linkage

(1) External linkage

External linkage refers to identifiers to be linked in translation (compiling) units that constitute the entire program and as a collection of libraries.

The following identifiers have external linkage examples:

- The identifier of a function declared without storage class specifier
- The identifier of an objects or function declared as extern, which has no storage class specification
- The identifier of an object which has file scope but has no storage class specification.

(2) Internal linkage

Internal linkage refers to identifiers to be linked within 1 translation (compiling) unit.

The following identifier has an internal linkage example:

- The identifier of an object or function which has file scope and contains the storage class specifier static.

(3) No linkage

An identifier that has no linkage to any other identifier is an inherent entity.

Examples of identifiers that have no linkage are as follows:

- An identifier which does not refer to a data object or function
- An identifier declared as a function parameter
- The identifier of an object which does not have storage class specifier extern inside a block

2.3.3 Name space for identifiers

All identifiers are classified into the following "name spaces":

Name Spaces	Expaination
Label name	Distinguished by a label declaration.
Tag name of structure, union, or enumeration	Distinguished by the keyword struct, union or enum
Member name of structure or union	Distinguished by the dot (.) operator or arrow (->) operator.
Ordinary identifiers (other than above)	Declared as ordinary declarators or enumeration type constants.

2.3.4 Storage duration of objects

Each object has a "storage duration" that determines its lifetime (how long it can remain in memory).

This storage duration is divided into the following 2 categories: Static storage duration and Automatic storage duration

(1) Static storage duration

Before executing an object program that has a static duration, an area is reserved for objects and values to be stored are initialized once.

The objects exist throughout the execution of the entire program and retain the values last stored.

Objects which have a static storage duration are as shown below.

- Objects which have external linkage
- Objects which have internal linkage
- Objects declared by storage class specifier static

(2) Automatic storage duration

For objects that have automatic storage duration, an area is reserved when they enter a block to be declared.

If initialization is specified, the objects are initialized as they enter from the beginning of the block. In this case, if any object enters the block by jumping to a label within the block, the object will not be initialized.

For objects that have automatic storage duration, the reserved area will not be guaranteed after the execution of the declared block.

Objects that have automatic storage duration are as follows:

- Objects which have no linkage
- Objects declared inside a block without storage class specifier static

2.4 Data Types

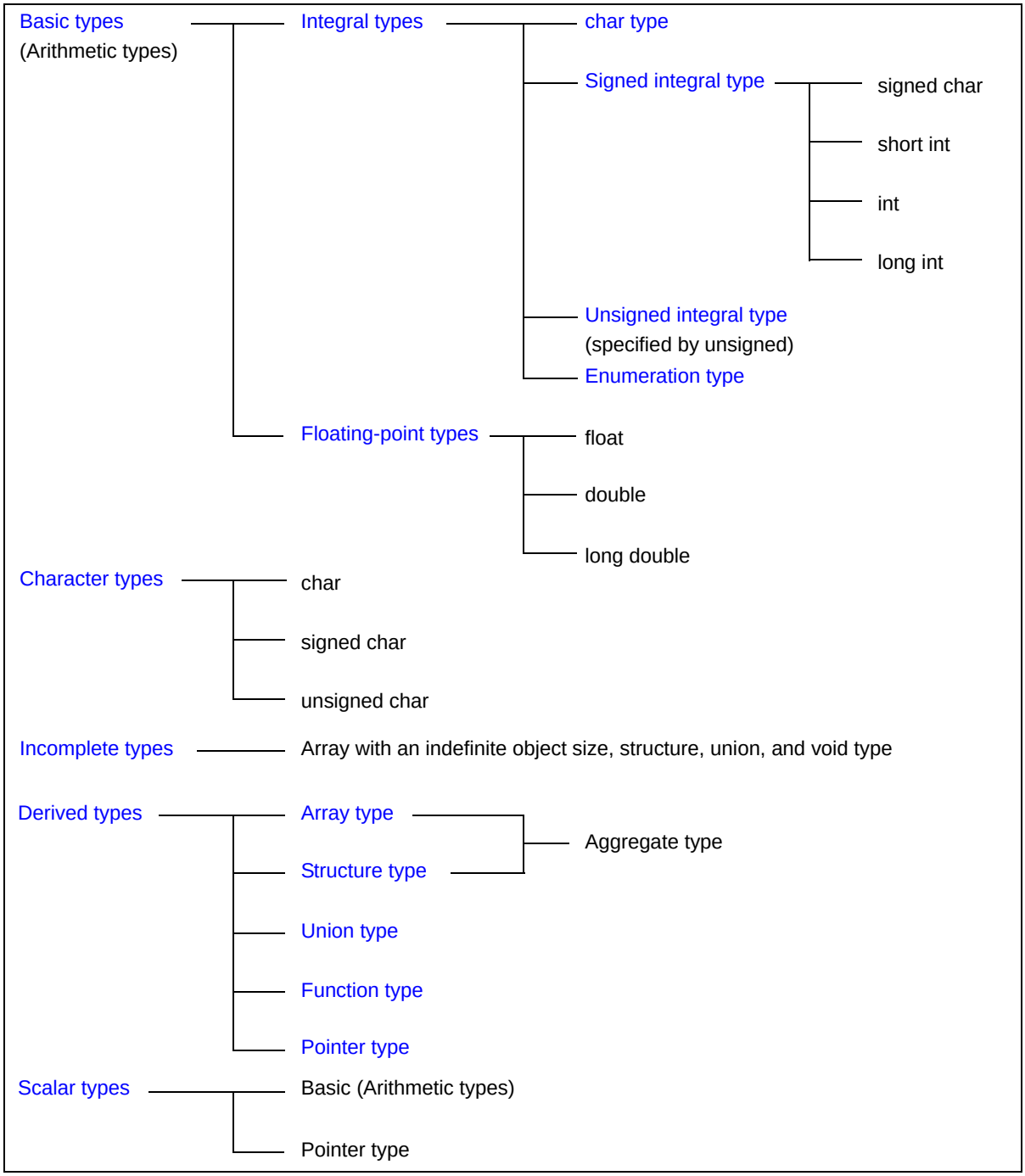
A type determines the meaning of a value to be stored in each object.

Data types are divided into the following 3 categories depending on the variable to be declared.

- Object type: Type which indicates an object with size information
- Function type: Type which indicates a function
- Incomplete type: Type which indicates an object without size information

Classification of types is shown below.

Figure 2-1 Classification of Types



2.4.1 Basic types

A collection of basic data types is also referred to as "arithmetic types".

The arithmetic types consist of integral types and floating-point type.

(1) Integral types

Integral data types are subdivided into 4 types. Each of these types has a value represented by the binary numbers 0 and 1.

- char type
- Signed integral type
- Unsigned integral type
- Enumeration type

(a) char type

The char type has a sufficient size to store any character in the basic execution character set.

The value of a character to be stored in a char type object becomes positive.

Data other than characters is handled as an unsigned integer.

In this case, however, if an overflow occurs, the overflowed part will be ignored.

(b) Signed integral type

The signed integral type is subdivided into the following 4 types:

- signed char
- short int
- int
- long int

An object declared with the signed char type has an area of the same size as the char type without qualifier.

An int object without qualifier has a size natural to the CPU architecture of the execution environment.

A signed integral type data has its corresponding unsigned integral type data.

Both share an area of the same size.

The positive number of a signed integral type data is a partial collection of unsigned integral type data.

(c) Unsigned integral type

The unsigned integral type is a data defined with the unsigned keyword. No overflow occurs in any computation involving unsigned integral type data. This is because of that if the result of a computation involving unsigned integral type data becomes a value which cannot be represented by an integral type, the value will be divided by the maximum number which can be represented by an unsigned integral type plus 1 and substituted with the remainder in the result of the division.

(d) Enumeration type

Enumeration is a collection or list of named integer constants.

An enumeration type consists of one or more sets of enumeration.

(2) Floating-point types

The floating-point types are subdivided into the 3 types.

- float
- double
- long double

In the CC78K0R, double and long double types as well as float type are supported as a floating-point expression for the single precision normalized number that is specified in ANSI/IEEE 754-1985. Thus, float, double, and long double types have the same value range.

Type	Value Range
(signed) char	-128 to +127
unsigned char	0 to 255
(signed) short int	-32,768 to +32,767
unsigned short int	0 to 65,535
(signed) int	-32,768 to +32,767
unsigned int	0 to 65,535
(signed) long int	-2,147,483,648 to +2,147,483,647
unsigned long int	0 to 4,294,967,295
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

Remark 1 The signed keyword can be omitted. However, with the char type, it is judged as signed char or unsigned char depending on the condition at the compilation time.

Remark 2 A short int data and an int data are handled as the data which have the same value range but are of the different types.

Remark 3 A unsigned short int data and an unsigned int data are handled as the data which have the same value range but are of the different types.

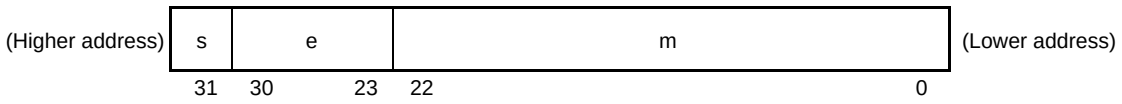
Remark 4 A float, double, and long double data are handled as the data which have the same value range but are of the different types.

Remark 5 The value ranges for float, double, and long double types are absolute.

[Floating-point number (float type) specifications]

(a) Format

The floating-point number format is shown below.



The 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	An exponent with a base of 2 is expressed as a 1-byte integer (expressed by 2's complement in the case of a negative), and used after having a further bias of 7FH added. These relationships are shown in the table below.
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.

Exponent (Hexadecimal)	Value of Exponent
FE	127
:	:
81	2
80	1
7F	0
7E	-1
:	:
01	-126

(b) Zero expression

When exponent = 0 and mantissa = 0, ±0 is expressed as follows.

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

(c) Infinity expression

When exponent = FFH and mantissa = 0, ±∞ is expressed as follows.

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

(d) Unnormalized value

When exponent = 0 and mantissa ≠ 0, the unnormalized 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 as is the 1st to 23rd decimal places.

(e) Not-a-number (NaN) expression

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

(f) Operation result rounding

Numerical values are rounded down to the nearest even number. If the operation 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 lowest binary bit is 0).

(g) Operation exceptions

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

Exception	Return Value
Underflow	Unnormalized number
Inexact	±0
Overflow	±∞
Zero division	±∞
Operation impossible	Not-a-number (NaN)

Calling the matherr function causes a warning to appear when an exception occurs.

2.4.2 Character types

The character data types include the following 3 types:

- char
- signed char
- unsigned char

2.4.3 Incomplete types

The incomplete data types include the following 4 types:

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

2.4.4 Derived types

The derived types are divided into the following 5 categories:

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

(1) Array type

The array type continuously allocates a collection of member objects called the element type.
Member objects all have an area of the same size. The array type specifies the number of element types and the elements of the array. It cannot create the array of incomplete type.

(2) Structure type

The structure type continuously allocates member objects each differing in size.
Giving it a name can specify each member object.

Remark The aggregate type is subdivided into 2 types:
Array type and Structure type. An aggregate type data is a collection of member objects to be taken successively.

(3) Union type

The union type is a collection of member objects that overlap each other in memory.
These member objects differ in size and name and can be specified individually.

(4) Function type

The function type represents a function that has a specified return value.

A function type data specifies the type of return value, the number of parameters, and the type of parameter.

If the type of return value is T, the function is referred to as a function that returns T.

(5) Pointer type

The pointer type is created from a function type object type called a referenced type as well as from an incomplete type.

The pointer type represents an object. The value indicated by the object is used to reference the entity of a referenced type.

A pointer type data created from the referenced type T is called a pointer to T.

2.4.5 Scalar types

The arithmetic types (basic type) and pointer type are collectively called the scalar types.

The scalar types include the following data types:

- char type
- Signed integral type
- Unsigned integral type
- Enumeration type
- Floating-point type
- Pointer type

2.4.6 Compatible type

If 2 types are the same, they are said to be compatible or have compatibility.

For example, if 2 structures, unions, or enumeration types that are declared in separate translation (compiling) units have the same number of members, the same member name and compatible member types, they have a compatible type. In this case, the individual members of the 2 structures or unions must be in the same order and the individual members (enumerated constants) of the 2 enumerated types must have the same values.

All declarations related to the same objects or functions must have a compatible type.

2.4.7 Composite type

A composite type is created from 2 compatible types. The following rules apply to the composite type.

- If either of the 2 types is an array of known type size, the composite type is an array of that size.
- If only one of the types is a function type which has a parameter type list (declared with a prototype), the composite type is a function prototype which has the parameter type list.
- If both types have a parameter type list (i.e., functions with prototypes), the composite type is one with a prototype consisting of all information that can be combined from the 2 prototypes.

<Example of composite type>

```
Assume that 2 declarations that have file scope are as follows:
        int f ( int ( * ) ( ) , double ( * ) [ 3 ] ) ;
        int f ( int ( * ) ( char * ) , double ( * ) [ ] ) ;
The composite type of the function in this case becomes as follows:
        int f ( int ( * ) ( char * ) , double ( * ) [ 3 ] ) ;
```

2.5 Constants

A constant is a variable, which does not change in value during the execution of the program, and its value must be set beforehand. A type for each constant is determined according to the format and value specified for the constant.

The following 4 constant types are available:

- Floating-point constants
- Integer constants
- Enumeration constants
- Character constants

2.5.1 Floating-point constant

A floating-point constant consists of an effective digit part, exponent part, and floating-point suffix.

Effective digit part	integer part, decimal point, and fraction part
Exponent part	e or E, signed exponent
Floating point suffix	f/F (float) l/L (long double) Remark If omitted (double)

The signed exponent of the exponent part and the floating-point suffix can be omitted.

Either the integer part or fraction part must be included in the effective digits. Also, either the decimal point or exponent part must be included (example: 1.23F, 2e3).

2.5.2 Integer constant

An integer constant starts with a number and does not have the decimal point nor exponent part.

An unsigned suffix can be added after the integer constant to indicate that the integer constant is unsigned. A long suffix can be added after the integer constant to indicate that the integer constant is long.

```
Unsigned suffix
        u  U
Long suffix
        l  L
```

There are the following 3 types of integer constant.

- Decimal constant
- Octal constant
- Hexadecimal constant

Details on each integer constant are as follows.

Table 2-7 Integer Constant

Integer Constant	Expaination
Decimal constant	A decimal constant is an integer value with the base (radix) of 10. [Specification] A decimal constant must begin with a number other than 0 followed by any numbers 0 through 9 (example: 56U). decimal number that starts with a number other than 0[Note] Note Decimal number = 1 2 3 4 5 6 7 8 9
Octal constant	An octal constant is an integer value with the base of 8. [Specification] An octal constant must begin with 0 followed by any numbers 0 through 7 (example: 034U). integer suffix 0 + octal number[Note] Note Octal number = 0 1 2 3 4 5 6 7
Hexadecimal constant	A hexadecimal constant is an integer value with the base of 16. [Specification] A hexadecimal constant must begin with 0x or 0X followed by any numbers 0 through 9 and a through f or A through F which represent 10 through 15 (example: 0xF3). integer suffix 0x or 0X + hexadecimal number[Note] Note Hexadecimal number = 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

The type of integer constant is regarded as the first of the "representable type" shown in the table below. In the CC78K0R, the type of the unsubscripted constant can be changed to char or unsigned char depending on the compile condition (option).

Table 2-8 Integer Constant and Representable Type

Integer Constant	Representable Type
Unsuffixed decimal number	int, long int, unsigned long int
Unsuffixed octal, hexadecimal number	int, unsiged int, long int, unsigned long int
Suffixed u or U	unsiged int, unsigned long int
Suffixed l or L	long int, unsigned long int
Suffixed u or U, and suffixed l or L	unsigned long int

2.5.3 Enumeration constants

Enumeration constants are used for indicating an element of an enumeration type variable, that is, the value of an enumeration type variable that can have only a specific value indicated by an identifier.

The enumeration type (enum) is whichever is the first type from the top of the list of 3 types shown below that can represent all the enumeration constants. The enumeration constant is indicated by the identifier.

- signed char
- unsigned char
- signed int

It is described as "enum enumeration type {list of enumeration constant}".

<Example>

```
enum    months { January = 1 , February , March , April , May } ;
```

When the integer is specified with =, the enumeration variable has the integer value, and the following value of enumeration variable has that integer value + 1. In the example shown above, the enumeration variable has 1, 2, 3, 4, 5, respectively. When there is not "= 1", each constant has 0, 1, 2, 3, 4, 5, respectively.

2.5.4 Character constants

A character constant is one or more character strings enclosed in a pair of single quotes as in ’X’ or ’ab’.

A character constant does not include single quote’, back slash (\), and line feed character (\n). To represent these characters, escape sequences are used.

There are the following 3 types of escape sequences.

- Simple escape sequence

  \’ \" \? \\ \a \b \f \n \r \t \v

- Octal escape sequence

  \octal number [octal number octal number]

  (example: \012, \0[Note 1])

- Hexadecimal escape sequence

  \x hexadecimal number

  (example: \xFF[Note 2])

Note 1 Null character

Note 2 In the CC78K0R, \xFF represents -1. If the condition (option) that regards char as unsigned char is added, however, it represents +255.

2.6 String Literal

A string literal is a string of zero or more characters enclosed in a pair of double quotes as in "xxx" (example: "xyz").

A single quote (’) is represented by the single quotation mark itself or by ESCAPE sequence \’, whereas a double quote (") is represented by ESCAPE sequence \".

Array elements have char type string literal and are initialized by tokens given (example: char array [ ] = "abc";).

2.7 Operators

The operators are shown in the table below.

Table 2-9 List of Operators

```
[ ]  ( )  .  ->
++  --  &  *  +  -  ~  !  sizeof
/  %  <<  >>  <  >  <=  >=  ==  !=
^  |  &&  ||
?  :
=  *=  /=  %=  +=  -=  <<=  >>=
&=  ^=  |=
,  #  ##
```

The [ ], ( ), and ?: operators must always be used in pairs.

An expression may be described in brackets "[ ]", in parentheses "( )", or between "?" and ":".

The # and ## operators are used only for defining macros in preprocessor directives. For the description, refer to "CHAPTER 5 OPERATORS AND EXPRESSIONS".

2.8 Delimiters

A delimiter is a symbol that has an independent syntax or meaning. However, it never generates a value. The following delimiters are available for use in C.

```
[ ]  ( )  { }  *  , :  =  ;  ...  #
```

In brackets "[ ]", parentheses "( )", or braces "{ }", an expression declaration or statement may be described. These delimiters must always be used in pairs as shown above. The delimiter "#" is used only for preprocessor directives.

2.9 Header Name

A header name indicates the name of an external source file. This name is used only in the preprocessor directive "#include".

An example of #include instruction of a header name is shown below. For the details of each #include instruction, refer to "9.2 Source File Inclusion Directive".

```
#include        <header-name>
#include        "header-name"
```

2.10 Comment

A comment refers to a statement to be included in a C source module for information only.

It begins with "/*" and ends with "*/".

The CC78K0R can identify multi-byte characters, including kanji (2-byte characters). The use of kanji code can be specified by using an option or the environment variable.The part after "//" to the line feed can be identified as a comment by the -zp option.

```
/* comment */
// comment
```

CHAPTER 3 DECLARATION OF TYPES AND STORAGE CLASSES

This chapter explains how data (variables) or functions to be used in C should be declared as well as scope for each data or function.

A declaration means the specification of an interpretation or attribute for an identifier or a collection of identifiers. A declaration to reserve a storage area for an object or function named by an identifier is referred to as a "definition".

An example of a declaration is shown below.

<Example of Declaration of Type and Storage Classes>

```
#define TRUE    1
#define FALSE   0
#define SIZE    200

void    main ( void ) {
                                    /* Declaration of automatic variables */
        auto    int     i , prime , k ;

        for ( i = 0 ; i <= SIZE ; i++ )
            mark [ i ] = TRUE ;
        :
}
```

A declaration is configured with storage class specifier, type specifier, initialize declarator, etc. The storage class specifier and type specifier specify the linkage, storage duration, and the type of an entity indicated by declarator. An initialize declarator list is a list of declarators each delimited with a comma. Each declarator may have additional type information or initializer or both.

If an identifier for an object is declared that it has no linkage, a type for the object must be perfect (the object with information related to the size) at the end of the declarator or initialize declarator (if it is with any initializer).

3.1 Storage Class Specifiers

A storage class specifier specifies the storage class of an object.

It indicates the storage location of a value, which the object has, and the scope of the object. In a declaration, only 1 storage class specifier can be described.

The following 5 storage class specifiers are available:

Table 3-1 Storage Class Specifiers

Type of Specifier	Meaning
typedef	The typedef specifier declares a synonym for the specified type. See "3.6 typedef Declarations" for details of the typedef specifier.
extern	The extern specifier indicates (tells the compiler) that a variable immediately before this specifier is declared elsewhere in the program (i.e., an external variable).
static	The static specifier indicates that an object has static storage duration. For an object, which has static storage duration, an area is reserved before the program execution and a value to be stored is initialized only once. The object exits throughout the execution of the entire program and retains the value last stored in it.
auto	The auto specifier indicates that an object has automatic storage duration. For an object that has automatic storage duration, an area is reserved when the object enters a block to be declared. At entry into the declared block from its top, the object is initialized if so specified. If the object enters the block by jumping to a label within the block, the object will not be initialized. The area reserved for an object, which has automatic storage duration, will not be guaranteed after the execution of the declared block.
register	The register specifier indicates that an object is assigned to a register of the CPU. With the CC78K0R, it is allocated to the register or saddr area of the CPU. See "CHAPTER 11 EXTENDED FUNCTIONS" for details of register variables.

3.2 Type Specifiers

A type specifier specifies (or refers to) the type of an object.

The following type specifiers are available:

- void
- char
- short
- int
- long
- float
- double
- long double
- signed
- unsigned
- Structure specifier and union specifier
- Enumeration specifiers
- typedef name

In the CC78K0R, the following type specifiers have been added.

```
bit/boolean/__boolean
```

The followings explain the meaning of each type specifier and the limit values that can be expressed with the CC78K0R (the values enclosed in the parentheses). Since the CC78K0R supports only the single precision of IEEE Std 754-1985 for floating-point operations, double and long double data are regarded to have the same format as those of float data.

Type specifiers separated from each other with a slash have the same size.

Table 3-2 Type Specifiers

Type of Specifier	Meaning	Limit
void	Collection of null values	-
char	Size of the basic character set that can be stored	-
signed char	Signed integer	-128 to +127
unsigned char	Unsigned integer	0 to 255
short/signed short/ short int /signed short int	Signed integer	-32,768 to +32,767
unsigned short/unsigned short int	Unsigned integer	0 to 65,535
int/signed/signed int	Signed integer	-32,768 to +32,767
unsigned/unsigned int	Unsigned integer	0 to 65,535
long/signed long/ long int/signed long int	Signed integer	-2,147,483,648 to +2,147,483,647
unsigned long/ unsigned long int	Unsigned integer	0 to 4,294,967,295
float	Single precision floating point number	1.17,549,435E - 38F to 3.40,282,347E + 38F[Note]
double	Double precision floating point number	1.17,549,435E - 38F to 3.40,282,347E + 38F[Note]
long double	Extended precision floating point number	1.17,549,435E - 38F to 3.40,282,347E + 38F[Note]
structure/union specifier	Collection of member objects	-
enumeration specifier	Collection of int type constants	-
typedef	Synonym of specified type	-
bit/boolean/__boolean	Integers represented with a single bit	0 to 1

Note Range of absolute values

3.2.1 Structure specifier and union specifier

Both the structure specifier and union specifier indicate a collection of named members (objects).

These member objects can have different types from one another.

(1) Structure specifier

The structure specifier declares a collection of two or more different types of variables as 1 object.

Each type of object is called a member and can be given a name. For members, continuous areas are reserved in the order of their declarations.

However, because the 78K0R contains a restriction whereby word data is unable to be read from or written to odd addresses, the code size is prioritized by default, and align data is inserted to ensure members of 2 bytes or more are allocated to even addresses. Gaps may therefore occur between members due to the align data.

The -rc option can be specified to inhibit insertion of align data and enable structures to be packed. In this case, although the size of the data is reduced, members of 2 or more bytes allocated to odd addresses are read/written using 1-byte unit read/write code, which increases the code size.

The structure is declared as follows. The declaration will not yet allocate memory since it does not have a list of structure variables. For the definition of the structure variables, refer to "CHAPTER 7 STRUCTURES AND UNIONS".

```
struct  identifier {
        member-declaration-list
} ;
```

<Example of structure declaration>

```
struct  tnode  {
        int     count ;
        struct  tnode   *left , *right ;
} ;
```

(2) Union specifier

The union specifier declares a collection of two or more different types of variables as 1 object. Each type of object is called a member and can be given a name. The members of a union overlay each other in area, namely, they share the same area.

The union declares as follows. The declaration will not yet allocate memory since it does not have a list of union variables. For the definition of the union variables, refer to "CHAPTER 7 STRUCTURES AND UNIONS".

```
union   identifier {
        member-declaration-list
} ;
```

<Example of union declaration>

```
union   u_tag  {
        int     var1 ;
        long    var2 ;
} ;
```

Each member object can be any type other than the incomplete types or function types. The member can declare with the number of bits specified. The member with the number of bits specified is called a bit field. In the CC78K0R, extended functions related to bit field declaration have been added. For the details, refer to "11.5 Bit field declaration".

(3) Bit field

A bit field is an integral type area consisting of a specified number of bits.

For the bit field, int type, unsigned int type, and signed int type data can be specified.[Note 1]

Whether the MSB of an int bit field which has no qualifier is judged as a sign bit differs depending on the microcontroller used. In some microcontrollers, a signed type bit field is handled as an unsigned type[Note 2].

If two or more bit fields exist, the second and subsequent bit fields are packed into the adjacent bit positions, provided there is an ample space within the same memory unit. By placing an unnamed bit field with a width of 0, the next bit field will not be packed into a space within the same memory unit. An unnamed bit field has no declarator and declares a colon and a width only.

Unary&operator (address) cannot be applied to the bit field object.

Note 1 In the CC78K0R, char type, unsigned char type, and signed char type can also be specified. All of them are regarded as unsigned type since the CC78K0R does not support signed type bit field.

Note 2 In the CC78K0R, the direction of bit field allocation can be changed by compiler option -rb (for the details, refer to "CHAPTER 11 EXTENDED FUNCTIONS").

<Example of bit field declaration>

```
struct  data {
        unsigned int    a : 2 ;
        unsigned int    b : 3 ;
        unsigned int    c : 1 ;
} no1 ;
```

3.2.2 Enumeration specifiers

An enumeration type specifier indicates a list of objects to be put in sequence.

Objects to be declared with the enum specifier will be declared as constants that have int types.

The enumeration specifier declares as shown below.

```
enum    identifier {
        enumerator-list
} ;
```

Objects are declared with an enumerator list.

Values are defined for all objects in the list in the order of their declaration by assigning the value of 0 to the first object and the value of the previous object plus 1 to the 2nd and subsequent objects. A constant value may also be specified with "=".

In the following example, "hue" is assumed as the tag name of the enumeration, "col" as an object that has this (enum) type, and "cp" as a pointer to an object of this type. In this declaration, the values of the enumeration become "{0, 1, 20, 21}".

```
enum    hue  {
        chartreuse ,
        burgundy ,
        claret = 20 ,
        winedark
} ;

enum    hue     col , *cp ;

void    main ( void ) {
        col = claret ;
        cp = &col ;

        if ( *cp != burgundy ) {
                :
        } else {
                :
        }
            :
}
```

3.2.3 Tags

A tag is a name given to a structure, union, or enumeration type.

A tag has a declared data type and objects of the same type can be declared with a tag.

An identifier in the following declaration is a tag name.

```
structure/union dentifier  { member-declaration-list } ;
or
enum            identifier { enumerator-list } ;
```

A tag has the contents of the structure/union or enumeration defined by a member. In the next and subsequent declarations, the structure of a struct, union, or enum type becomes the same as that of the tag's list. In the subsequent declarations within the same scope, the list enclosed in braces must be omitted. The following type specifier is undefined with respect to its contents and thus the structure or union has an incomplete type.

```
structure/union identifier ;
```

A tag to specify the type of this type specifier can be used only when the object size is unnecessary. This is because of that by defining the contents of the tag within the same scope, the type specification becomes incomplete.

In the following example, the tag "tnode" specifies a structure that includes pointers to an integer and 2 objects of the same type.

```
struct  tnode {
        int     count ;
        struct  tnode   *left , *right ;
} ;
```

The next example declares "s" as an object of the type indicated by the tag (tnode) and "sp" as a pointer to the object of the type indicated by the tag. By this declaration, the expression "sp -> left" indicates a pointer to "struct tnode" on the left of the object pointed to by "sp" and the expression "s.right -> count" indicates "count" which is a member of "struct tnode" on the right of "s".

```
typedef struct  tnode   TNODE ;
struct  tnode {
        int     count ;
        struct  tnode   *left , *right ;
} ;

TNODE   s , *sp ;

void    main ( void ) {
        sp -> left = sp -> right ;
        s.right -> count = 2 ;
}
```

3.3 Type Qualifiers

2 type qualifiers are available: const and volatile.

These type qualifiers affect Lvalues only.

Using an Lvalue that has non-const type qualifier cannot change an object that has been defined with const type qualifier. Using an Lvalue that has non-volatile type qualifier cannot reference an object that has been defined with volatile type qualifier.

An object that has volatile qualifier type can be changed by a method not recognizable by the compiler or may have other unnoticeable side effects. Therefore, an expression that references this object must be strictly evaluated according to the sequence rules that regulate abstractly how programs written in C should be executed. In addition, the values to be last stored in the object at every sequence point must be in agreement with those determined by the program except the changes due to the factors unrecognizable by the compiler as mentioned above.

If an array type is specified with type qualifiers, the qualifiers apply to the array members, not the array itself.

No type qualifier can be included in the specification of a function type. However, callt, __callt, callf, __callf, noauto, norec, __leaf, __interrupt, __interrupt_brk, __rtos_interrupt, __near, __far, which are the type qualifiers unique to the CC78K0R mentioned in "2.2 Keywords", can be included as type qualifiers.

sreg, __sreg, __directmap, __temp, __near, __far are also type qualifiers.

In the following example, "real_time_clock" can be changed by hardware, but such operations as assignment, increment, and decrement are not allowed in.

```
extern  const   volatile        int     real_time_clock ;
```

An example of modifying aggregate type data with type qualifiers is shown below.

```
const   struct  s  { int mem ; } cs = { 1 } ;
struct  s       ncs ;                           /* Object ncs is changeable */
typedef int     A [ 2 ] [ 3 ] ;
const   A a = { { 4 , 5 , 6 } , { 7 , 8 , 9 } } ; /* Array of const int array */
int     *pi ;
const   int     *pci ;

ncs = cs ;                  /* Correct */
cs = ncs ;
                            /* Violates restriction of Lvalue which has */
                            /* modifiable assignment operator */
pi = &ncs.mem ;             /* Correct */
pi = &cs.mem ;              /* Violates restriction of the type of assignment */
                            /* operator = */
pci = &cs.mem ;             /* Correct */
pi = a [ 0 ] ;              /* Incorrect: a [ 0 ] has "const int *" type */
```

3.4 Declarators

A declarator declares an identifier.

Here, pointer declarators, array declarators, and function declarators are mainly discussed.

By a declarator, the scope of an identifier and a function or object which has a storage duration and a type are determined.

The description of each declarator is shown below.

3.4.1 Pointer declarators

A pointer declarator indicates that an identifier to be declared is a pointer.

A pointer points to (indicates) the location where a value is stored. Pointer declarations are performed as follows.

```
*type qualifier-list    identifier ;
```

By this declaration, the identifier becomes a pointer to T1.

The following 2 declarations indicate a variable pointer to a constant value and an invariable pointer to a variable value, respectively.

```
const   int     *ptr_to_constant ;
int     *const  constant_ptr ;
```

The first declaration indicates that the value of the constant "const int" pointed by the pointer "ptr_to_constant" cannot be changed, but the pointer "ptr_to_constant" itself may be changed to point to another "const int". Likewise, the second declaration indicates that the value of the variable "int" pointed by the pointer "constant_ptr" may be changed, but the pointer "constant_ptr" itself must always point to the same position.

The declaration of the invariable pointer "constant_ptr" can be made distinct by including a definition for the pointer type to the int type data.

The following example declares "constant_ptr" as an object that has a const qualifier pointer type to int.

```
typedef int     *int_ptr ;
const   int_ptr constant_ptr ;
```

3.4.2 Array declarators

An array declarator declares to the compiler that an identifier to be declared is an object that has an array type.

Array declaration is performed as shown below.

```
type    identifier      [constant-expression] ;
```

By this declaration, the identifier becomes an array that has the declared type. The value of the constant expression becomes the number of elements in the array. The constant expression must be an integer constant expression which has a value greater than 0. In the declaration of an array, if a constant expression is not specified, the array becomes an incomplete type.

In the following example, a char type array "a[ ]" which consists of 11 elements and a char type pointer array "ap[ ]" which consists of 17 elements have been declared.

```
char    a [ 11 ] , *ap [ 17 ] ;
```

In the following 2 examples of declarations, "x" in the first declaration specifies a pointer to an int type data and "y" in the second declaration specifies an array to an int type data which has no size specification and is to be declared elsewhere in the program.

```
extern  int     *x ;
extern  int     y [ ] ;
```

3.4.3 Function declarators (including prototype declarations)

A function declarator declares the type of return value, argument, and the type of the argument value of a function to be referenced.

Function declaration is performed as follows.

```
type    identifier      (parameter-list or identifier-list) ;
```

By this declaration, the identifier becomes a function which has the parameter specified by the parameter type list and returns the value of the type declared before the identifier. Parameters of a function are specified by a parameter identifier lists. By these lists, an identifier, which indicates argument and its type, are specified. A macro defined in the header file "stdarg.h" converts the list described by the ellipsis (, ...) into parameters. For a function that has no parameter specification, the parameter list will become "void".

3.5 Type Names

A type name is the name of a data type that indicates the size of a function or object.

Syntax-wise, it is a function or object declaration less identifiers.

Examples of type names are given below.

Table 3-3 Examples of Type Names

Examples of Type Names	Explain
int	Specifies an int type.
int *	Specifies a pointer to an int type.
int * [3]	Specifies an array which has 3 pointers to an int type.
int (*) [3]	Specifies a pointer to an array which has 3 int types.
int * ( )	Specifies a function which returns a pointer to an int type which has no parameter specification.
int (*) (void)	Specifies a pointer to a function which returns an int type which no parameter specification.
int (*const [ ]) (unsigned int, ...)	Specifies an indefinite number of arrays which have 1 parameter of unsigned int type and an invariable pointer to each function that returns an int type.

3.6 typedef Declarations

The typedef keyword defines that an identifier is a synonym to a specified type. The defined identifier becomes a typedef name.

The syntax of typedef names is shown below.

```
typedef type    identifier ;
```

In the following example, "distance" is an int type, the type of "metricp" is a pointer to a function that returns an int type that has no parameter specification, the type of "z" is a specified structure, and "zp" is a pointer to this structure.

```
typedef int     MILES , KLICKSP ( ) ;
typedef struct  { long re , im ; }      complex ;
  :
MILES   distance ;
extern  KLICKSP *metricp ;
complex z , *zp ;
```

In the following example, typedef name t is declared with signed int type, and typedef name plain is declared with int type, respectively, and the structure with 3 bit field members is declared. The bit field members are as follows.

- Bit field member with name t and the value 0 to 15
- Bit field member without a name and the const qualified value -16 to +15 (if accessed)
- Bit field member with name r and the value -16 to +15

```
typedef signed  int     t ;
typedef int     plain ;

struct  tag {
        unsigned        t : 4 ;
        const           t : 5 ;
        plain           r : 5 ;
} ;
```

In this example, these 2 bit field declarations differ in the point that the first bit field declaration has unsigned as the type specifier (therefore, t becomes the name of the structure member), and the second bit field declaration, on the other hand, has const as the type qualifier (qualifiers t which can be referred to as typedef name). After this declaration, if the following description is found within the effective range, the function f is declared as "function which has 1 parameter and returns signed int", and the parameter is declared as "pointer type for the function which has 1 parameter and returns signed int". The identifier t is declared as long type.

```
t       f ( t ( t ) ) ;
long    t ;
```

typedef names may be used to facilitate program reading. For example, the following 3 declarations for the function signal all specify the same type as the first declaration which does not use typedef.

```
typedef void    fv ( int ) ;
typedef void    ( *pfv ) ( int ) ;

void    ( *signal ( int , void ( * ) ( int ) ) ) ( int ) ;
fv      *signal ( int , fv * ) ;
pfv     signal ( int , pfv ) ;
```

3.7 Initialization

Initialization refers to setting a value in an object beforehand.

An initializer carries out the initialization of an object.

Initialization is performed as follows.

```
object = { initializer-list } ;
```

An initializer list must contain initializers for the number of objects to be initialized.

All expressions in initializers or an initializer list for objects that have static storage duration and objects that have an aggregate type or a union type must be specified with constant expressions.

Identifiers that declare block scope but have external or internal linkage cannot be initialized.

3.7.1 Initialization of objects which have a static storage duration

If no attempt is made to initialize an arithmetic type object that has static storage duration, the value of the object will be implicitly initialized to 0.

Likewise, a pointer type object which has a static storage duration will be initialized to a null pointer constant.

<Example>

```
unsigned    int     gval1 ;         /* Initialized by 0 */
static      int     gval2 ;         /* Initialized by 0 */
void        func ( void ) {
            static  char    aval ;  /* Initialized by 0 */
}
```

3.7.2 Initialization of objects which have an automatic storage duration

The value of an object which has an automatic storage duration becomes indefinite and will not be guaranteed if it is not initialized.

<Example>

```
void    func ( void ) {
        char    aval ;  /* Undefined value at this point */
            :
        aval = 1 ;      /* Initialized by 1 */
}
```

3.7.3 Initialization of character arrays

A char character array can be initialized with char string literal (char string enclosed with " "). Likewise, a character string in which a series of char string literal are contained initializes the individual members or elements of an array.

- In the following example, the array objects s and t with "no type qualifier" are defined and the elements of each array will be initialized by char string literal.

```
char    s [ ] = "abc" , t [ 3 ] = "abc" ;
```

- The next example is the same as the above example of array initialization.

```
char    s [ ] = { 'a' , 'b' , 'c' , '\0' } ,
        t [ ] = { 'a' , 'b' , 'c' } ;
```

- The next example defines p as "pointer to char" type and the member is initialized by characteristic string literal so that length indicates "char array" type object.

```
char    *p = "abc" ;
```

3.7.4 Initialization of aggregate or union type objects

(1) Aggregate type

An aggregate type object is initialized with a list of initializers described in ascending order of subscripts or members. The initializer list to be specified must be enclosed in braces.

If the number of initializers in the list is less than the number of aggregate members, the members not covered by the initializers will be implicitly initialized just the same as an object which has a static storage duration.

With an array with an unknown size, the number of its elements is governed by the number of initializers and the array will no longer become an incomplete type.

(2) Union type

A union type object is initialized with an initializer for the first member of the union that is enclosed in braces.

- In the following example, the array "x" with an unknown size will change to a 1-dimensional array that has 3 elements as a result of its initialization.

```
int     x [ ] = { 1 , 3 , 5 } ;
```

- The next example shows a complete definition which has initializers enclosed in braces.
"{1, 3, 5}" initializes "y [0] [0]", "y [0] [1]", and "y [0] [2]" in the 1st line of the array object "y [0]". Likewise, in the second line, the elements of the array objects "y [1]" and "y [2]" are initialized. The initial value of "y [3]" is 0 since it is not specified.

```
char    y [ 4 ] [ 3 ] = {
        { 1 , 3 , 5 } ,
        { 2 , 4 , 6 } ,
        { 3 , 5 , 7 }
} ;
```

- The next example produces the same result as the above example.

```
char    z [ 4 ] [ 3 ] = {
        1 , 3 , 5 , 2 , 4 , 6 , 3 , 5 , 7
} ;
```

- In the following example, the elements in the first row of "z" are initialized to the specified values and the rest of the elements are initialized to 0.

```
char    z [ 4 ] [ 3 ] = {
        { 1 } , { 2 } , { 3 } , { 4 }
} ;
```

- In the next example, a 3-dimensional array is initialized.

  q [0] [0] [0] are initialized to 1, q [1] [0] [0] to 2, and q [1] [0] [1] to 3. 4, 5 and 6 initialize q [2] [0] [0], q [2] [0] [1], and q [2] [1] [0], respectively. The rest of the elements are all initialized to 0.

```
short   q [ 4 ] [ 3 ] [ 2 ] = {
        { 1 } ,
        { 2 , 3 } ,
        { 4 , 5 , 6 }
} ;
```

- The following example produces the same result as the above initialization of the 3-dimensional array.

```
short   q [ 4 ] [ 3 ] [ 2 ] = {
        1 , 0 , 0 , 0 , 0 , 0 ,
        2 , 3 , 0 , 0 , 0 , 0 ,
        4 , 5 , 6
} ;
```

- The following example shows a complete definition of the above initialization using braces.

```
short   q [ 4 ] [ 3 ] [ 2 ] = {
        {
                { 1 } ,
        } ,
        {
                { 2 , 3 } ,
        } ,
        {
                { 4 , 5 , 6 } ,
        }
} ;
```

CHAPTER 4 TYPE CONVERSIONS

In an expression, if 2 operands differ in data type, the compiler automatically performs a type conversion operation. This conversion is similar to a change obtained by the cast operator. This automatic type conversion is called an implicit type conversion. In this chapter, this implicit type conversion is explained.

Type conversion operations include usual arithmetic conversions, conversions involving truncation/round off, and conversions involving sign change. A list of conversions between types is shown in the table below.

Table 4-1 List of Conversions Between Types

Before Conversion		After Conversion										
		(signed) char	unsigned char	(signed) short int	unsigned short int	(signed) int	unsigned int	(signed) long int	unsigned long int	float	double	long double
(signed) char	+	\	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
	-	\	NG	OK	NG	OK	NG	OK	NG	OK	OK	OK
unsigned char		Δ	\	OK	OK	OK	OK	OK	OK	OK	OK	OK
(signed) short int	+			\	OK	\	OK	OK	OK	OK	OK	OK
	-			\	NG	\	NG	OK	NG	OK	OK	OK
unsigned short int				Δ	\	Δ	\	OK	OK	OK	OK	OK
(signed) int	+			\	OK	\	OK	OK	OK	OK	OK	OK
	-			\	NG	\	NG	OK	NG	OK	OK	OK
unsigned int				Δ	\	Δ	\	OK	OK	OK	OK	OK
(signed) long int	+							\	OK	OK	OK	OK
	-							\	NG	OK	OK	OK
unsigned long int								Δ	\	OK	OK	OK
float										\	OK	OK
double											\	OK
long double												\

OK: Type conversion will be performed properly.

\: Type conversion will not be performed.

NG: A correct value will not be generated. (The data type will be regarded as an unsigned int type.)

Δ: The data type will not change bit-image-wise. However, if a positive number cannot represent it sufficiently, no correct value will be generated. (regarded as an unsigned integer)

Blank: An overflow in the result of the conversion will be truncated.
The + or - sign of the data may be changed depending on the type after the conversion.

Remark The signed keyword can be omitted. However, with a char type data, the data type is regarded as the signed char or unsigned char type depending on the compile-time condition (option).

4.1 Arithmetic Operands

(1) Characters and integers (general integral promotion)

The data types of char, short int, and int bit fields (whether they are signed or unsigned) or of objects that have an enumeration type will be converted to int types if their values are within the range that can be represented with int types. If not within the range, they will be converted to unsigned int types. These implicit type conversions are referred to as "general integral general promotion".

All other arithmetic types will not be changed by this general integral promotion. General integral promotion will retain the value of the original data type including its sign.

char type data without type qualifier will normally be handled as signed char in the CC78K0R. It can be handled as an unsigned char with option.

(2) Signed integers and unsigned integers

When a value with an integer type is converted to another, the value will not be changed if the value can be expressed with the integer type after conversion.

When a signed integer is converted to an unsigned integer of the same or larger size, the value is not changed unless the value of the signed integer is negative. If the value of the signed integer is negative and the unsigned integer has a size larger than that of the signed integer, the signed integer is expanded to the signed integer with the same size as the unsigned integer, and then it is added with the value equal to the maximum number that can be expressed with the unsigned integer plus 1, and the signed integer before conversion is converted to the unsigned value.

When a value with an integer type is converted to an unsigned integer with a smaller size, the conversion result is a non-negative remainder which the value is divided with that value which 1 is added to the maximum number that can be expressed with an unsigned integer after conversion. When a value with an integer type is converted to a signed integer with smaller size or when an unsigned integer is converted to a signed integer with the same size, the overflown value is ignored if the value after conversion cannot be expressed. For the conversion pattern, refer to Table 4-1.

Conversion operations from signed integral type to unsigned integral type are as listed in the table below.

		unsigned	
		Smaller in Value Range	Greater in Value Range
signed	+	/	OK
	-	/	+

OK: Type conversion will be performed properly.

+: The data will be converted to a positive integer.

/: The result of the conversion will be the remainder of the integer value, modulo the largest possible value of the type to be converted plus 1.

(3) Usual arithmetic type conversions

Types obtained as a result of operations on arithmetic type data will have a wide range of values.

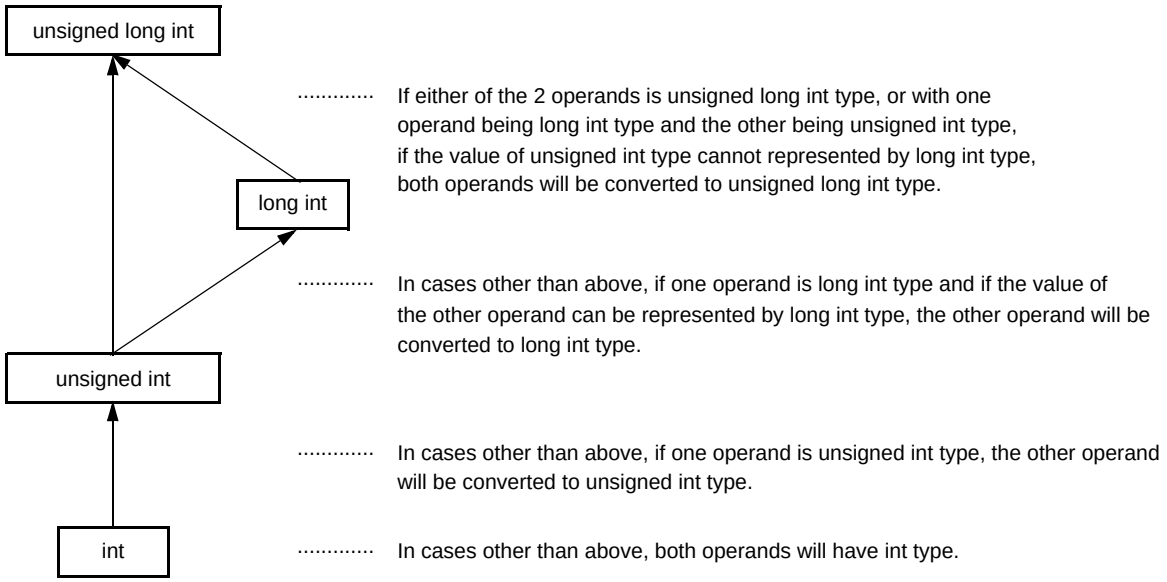
The type conversion of the operation result is performed as follows.

- If either one of the operands has long double type, the other operand is converted to long double type.
- If either one of the operands has double type, the other operand is converted to double type.
- If either one of the operands has float type, the other operand is converted to float type.

In cases other than above, general integer expansion is performed for both operands according to the following rules.

Figure 4-1 shows the rules.

Figure 4-1 Usual Arithmetic Type Conversions



In the CC78K0R, the conversion to int type can be intentionally disabled by compile condition (optimizing option).

For the details, refer to the CC78K0R C Compiler Operation User's Manual.

4.2 Other Operands

(1) Lvalues and function locators

An "Lvalue" refers to an expression that specifies an object (and has an incomplete type other than object type or void type).

Lvalues which do not have array types, incomplete types, or const qualifier types, and structures or unions which have no const qualifier type members are "modifiable Lvalues".

An Lvalue which has no array type will be converted to a value stored in the object to be specified, except when it is the operand of the sizeof operator, unary & operator, ++ operator, or -- operator or the left operand of an operator or an assignment operator. By being converted, it will no longer serve as an Lvalue.

The behaviors of Lvalues that have incomplete types but have no array types will not be guaranteed.

An Lvalue which has a "... array" type except character arrays will be converted to an expression which has a "pointer to ..." type. This expression is no longer an Lvalue.

A function locator is an expression that has a function type. With the exception of the operand of the sizeof operator or unary & operator, a function locator that has a "function type that returns ..." will be converted to an expression that has a "pointer type to a function that returns ...".

(2) void

The value (non-existent) of a void expression (i.e., an expression that has the void type) cannot be used in any way. Neither implicit nor explicit conversion to exclude void will be applied to this expression. If an expression of another type appears in the context which requires a void expression, the value of the expression or specifier is assumed to be non-existent.

(3) Pointers

A void pointer can be converted to a pointer to any incomplete type or object type. Conversely, a pointer to any incomplete type or object type can be converted to a void pointer. In either case, the result value must be equal to that of the original pointer.

An integer constant expression which has the value of 0 and has been cast to the void * type is referred to as a "null pointer constant". If the null pointer constant is substituted with, equal to, or compared with some pointer, the null pointer constant will be converted to that pointer.

CHAPTER 5 OPERATORS AND EXPRESSIONS

This chapter describes the operators and expressions to be used in the C language.

C has an abundance of operators for arithmetic, logical, and other operations. This rich set of operators also includes those for bit and address operations.

An expression is a string or combination of an operator and one or more operands. The operator defines the action to be performed on the operand(s) such as computation of a value, instructions on an object or function, generation of side effects, or a combination of these.

Examples of operators are given below.

```
#define TRUE    1
#define FALSE   0
#define SIZE    200

void    lprintf ( char * , int ) ;
void    putchar ( char c ) ;
char    mark [ SIZE + 1 ] ;                     /* + :  Arithmetic operator */

void    main ( void )  {
        int     i , prime , k , count ;

        count  = 0 ;                            /* = :  Assignment operator */
        for ( i = 0 ; i <= SIZE ; i++ )         /* ++ : Postfix operator */
            mark [ i ] = TRUE ;                 /* <= : Relational operator */

        for ( i = 0 ; i <= SIZE ; i++ ) {
            if ( mark [ i ] ) {
                prime = i + i + 3 ;             /* + :  Arithmetic operator */
                lprintf ( "%d" , prime ) ;
                count++ ;                       /* ++ : Postfix operator */
                if ( ( count%8 ) == 0 )         /* == : Relational operator */
                    putchar ( '\n' ) ;
                                                 /* += : Assignment operator */
                for ( k = i + prime ; k <= SIZE ; k += prime )
                    mark [ k ] = FALSE ;
            }
        }

        lprintf ( "Total %d\n" , count ) ;
loop1 : ;
        goto    loop1 ;
}

lprintf ( char *s , int i ) {
        int     j ;
        char    *ss ;
        j = i ;
        ss = s ;
}

void    putchar ( char c )  {
        char    d ;
        d = c ;
}
```

The table below shows the evaluation precedence of operators used in C.

Table 5-1 Evaluation Precedence of Operators

Type of Expression	Operator	Linkage[Note]	Priority
Postfix	[ ], ( ), ., ->, ++, --	-->	Highest
Unary	++, --, &, *, +, -, ~, !, sizeof	<--	
Cast	(type)	<--	
Multiplicative	*, /, %	-->	
Additive	+, -	-->	
Bitwise shift	<<, >>	-->	
Relational	<, >, <=, >=	-->	
Equality	==, !=	-->	
Bitwise AND	&	-->	
Bitwise XOR	^	-->	
Bitwise OR	|	-->	
Logical AND	&&	-->	
Logical OR	||	-->	
Conditional	? :	<--	
Assignment	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=	<--	
Comma	,	-->	Lowest

Note Operations in the same line contain the same priority.

The arrow (<-- or -->) in the "LINKAGE" column denotes that when an expression contains two or more operators in the same precedence, the operations are carried out in the direction of the arrow "-->" (from left to right) or "<--" (from right to left).

5.1 Primary Expressions

Primary expressions include the following:

- Identifier declared as an object or function
  (identifier primary expression)
- Constant (constant primary expression)
- String literal (constant primary expression)
- Expression enclosed in parentheses
  (parenthesized expression)

An identifier which becomes a primary expression is an Lvalue if an object is declared or a function locator if a function is declared. The data type of a constant is determined according to the value specified for the constant as explained in "2.5 Constants". String literal(s) become an Lvalue that has a data type as explained in "2.6 String Literal".

5.2 Postfix Operators

A postfix operator is an operator that appears or is placed after an object or a function.

The types of postfix operators are given below.

- Subscript operator
- Function call operator
- Structure and union member (.)
- Structure and union member (->)
- Postfix increment operator (++)
- Postfix decrement operator (--)

Subscript operator

SYNTAX

```
postfix-expression [subscripted-expression]
```

FUNCTION

- The [ ] subscript operator specifies or refers to a single member of an array object.

  The array or expression "E1 [E2]" is evaluated as if it were "(*(E1 + (E2)))". In other words, the value of E1 is a pointer to the first member of the array and E2 (if it is an integer) indicates the E2th member of E1 (counting from 0). With a multidimensional array, subscript operators as many as the number of dimensions must be connected.

  In the following example, x becomes an int type array of 3 * 5. In other words, x is an array which has 3 members each consisting of 5 int type members.

```
int     x [ 3 ] [ 5 ] ;
```

  A multidimensional array may be specified by connecting subscript operators.

  Assuming that E is an array of nth dimension (where n ≥ 2) consisting of i * j * ... * k, the array can be specified with the n number of subscript operators. In this case, E becomes a pointer to an array of (n - 1)th dimension consisting of j * ... * k.

NOTE

- A postfix expression must have a "... pointer to object". The subscripted expression of an array must be specified with integral type data. The result of the expression will become "..." type.

Function call operator

SYNTAX

```
postfix-expression (argument-expression-list) ;
```

FUNCTION

- The postfix "( )" operator calls a function.
  The function to be called is specified with a postfix expression and argument(s) to passed to the function are indicated in parentheses ( ).
- The description related to function includes the function prototype declaration, the function definition (the body of a function), and the function call. The function prototype declaration specifies the value a function returns, the type of argument, and the storage class.
- If the function prototype declaration is not referred to in a function call, each argument is extended with general integer. This is called "default actual argument extension". Performing a function prototype declaration avoids default actual argument extension and detects the mistakes of the type and number of argument and the type of return value.
- Calling a function which has neither storage class specification nor data type specification such as "identifier ( );" is interpreted as calling a function which has an external object and returns an int type which has no information on arguments. In other words, the following declaration will be made implicitly:

```
extern  int     identifier ( ) ;
```

[Example of function call]

```
int     func ( char , int ) ;          /* function prototype declaration */
char    a ;
int     b , ret ;

void    main ( void ) {
        ret = func ( a , b ) ;         /* function call */
}

int     func ( char c , int i )  {     /* function definition */
            :
        return  i ;
}
```

NOTE

- A function that returns an object other than array types can be called with this operator. The postfix expression must be of a pointer type to this function.
- In a function call including prototype, the type of argument must be of a type that can be assigned to the corresponding parameter(s). The number of arguments must also be in agreement.

Structure and union member (.)

SYNTAX

```
postfix-expression.identifier
```

FUNCTION

- The "." (dot) operator (also called a member operator) specifies the individual members of a structure or union.

  The postfix expression is the name of the structure or union object to be specified, and the identifier is the name of the member.

Structure and union member (->)

SYNTAX

```
postfix-expression -> identifier
```

FUNCTION

- The "->" (arrow) operator (also called an indirect membership operator) specifies the individual members of a structure or union.

  The postfix expression is the name of the pointer to the structure or union object to be specified, and the identifier is the name of the member.

<Examples of ".", "->" operators>

```
#include <stdlib.h>

union {
        struct  {
                int     type ;
        } n ;
        struct  {
                int     type ;
                int     intnode ;
        } ni ;
        struct  {
                int     type ;
                struct  {
                        long    longnode ;
                } *nl_p ;
        } nl ;
} u ;

void    func ( void )  {
        u.nl.type = 1 ;
        u.nl.nl_p -> longnode = -31415L ;
          :
        if ( u.n.type == 1 )
                u.nl.nl_p -> longnode = labs ( u.nl.nl_p -> longnode ) ;
}
```

Postfix increment operator (++)

SYNTAX

```
postfix-expression++
```

FUNCTION

- The postfix ++ (Increment) operator increments the value of an object by 1.

  This increment operation is performed by taking the data type of the object into account.

Postfix decrement operator (--)

SYNTAX

```
postfix-expression--
```

FUNCTION

- The postfix -- (Decrement) operator decrements the value of an object by 1.
  This decrement operation is performed by taking the data type of the object into account.

NOTE

- The operand of the postfix increment or decrement operator must be a modifiable Lvalue (qualified or unqualified).

5.3 Unary Operators

A unary operator performs an operation on 1 object or parameter (i.e., operand).

The following unary operators are available:

- Prefix increment operator (++)
- Prefix decrement operator (--)
- Unary & operator (&)
- Unary * operator (*)
- Unary arithmetic operators (+ - ~ !)
- sizeof operator

Prefix increment operator (++)

SYNTAX

```
++unary-expression
```

FUNCTION

- The prefix (Increment) operator increments the value of an object by 1.

  The expression "++E" of the prefix increment operator will produce the same result as the following expression.

```
E = E + 1
    or
E += 1
```

Prefix decrement operator (--)

SYNTAX

```
--unary-expression
```

FUNCTION

- The prefix -- (Decrement) operator decrements the value of an object by 1.

  The expression "--E" of the prefix decrement operator will produce the same result as the following expression:

```
E = E - 1
   or
E -= 1
```

Unary & operator (&)

SYNTAX

```
&operand
```

FUNCTION

- The unary & (address) operator returns the pointer of a specified object (i.e., the address of the variable it precedes).

Unary * operator (*)

SYNTAX

```
*operand
```

FUNCTION

- The unary * (indirection) operator returns the value indicated by a specified pointer (i.e., takes the value of the variable it precedes and uses that value as the address of the information in memory).

NOTE

- The operand of the unary & operator must be an Lvalue referring to an object not declared with the register storage class specifier. Neither a function locator nor a bit field can be used as the operand of this unary operator.

  The operand of the unary * operator must have a pointer type.

Unary arithmetic operators (+ - ~ !)

SYNTAX

```
+operand
-operand
~operand
!operand
```

FUNCTIONS

- The + (unary plus) operator performs positive integral promotion on its operand.
- The - (unary minus) operator performs negative integral promotion on its operand.
- The ~ (tilde) operator is a bitwise one's complement operator which inverts all the bits in a byte of its operand.
- The ! NOT or logical negation operator returns "0" if its operand is "0" and "1" if it is not "0". In other words, the operator changes each "0" to "1" and "1" to "0".

sizeof operator

SYNTAX

```
sizeof unary-expression
sizeof (type-name)
```

FUNCTION

- The sizeof operator returns the size of a specified object in bytes.

  The return value is governed by the data type of the object and the value of the object itself is not evaluated.

- The value to be returned by an unsigned char or signed char object (including its qualified type) on which a sizeof operation is performed is 1. With an array type object, the return value will be the total number of bytes in the array. With a structure or union type object, the result value will be the total number of bytes that the object would occupy including bytes necessary to pad out to the next appropriate alignment boundary.

- The type of the sizeof operation result is an integral type and its name is size_t. This name is defined in the <stddef.h> header. The sizeof operator is used mainly to allocate memory areas and transfer data to/from the I/O system.

EXAMPLE

- The following example finds the number of elements of an array by dividing the total number of bytes in the array by the size of a single element. Num becomes 5.

```
int     num ;
char    array [ ] = { 0 , 1 , 2 , 3 , 4 } ;

void    func ( void ) {
        num = sizeof array / sizeof array [ 0 ] ;
}
```

NOTE

- An expression that has a function type or incomplete type and an Lvalue which refers to a bit field object cannot be used as the operand of this operator.

5.4 Cast Operator

A cast is a special operator which forces one data type to be converted into another.

The cast operator is mainly used when converting a pointer type.

- Cast operator (type-name)

Cast operator (type-name)

SYNTAX

```
(type-name) expression
```

FUNCTION

- The cast operator converts the data type of another object (or the result of another expression) into the type specified in parentheses ( ).

EXAMPLE

```
void    func ( void )  {
        int     val ;
        float   f ;

        f = 3.14F ;
        val = ( int ) f ;               /* val becomes 3 by cast */
        val = * ( int * ) 0x10000 ;     /* Cast constant */
}
```

5.5 Arithmetic Operators

- * operator
- / operator
- % operator
- + operator
- - operator

Arithmetic operators are divided into multiplying operators and adding operators.

Multiplying operators find the product, quotient, and remainder of 2 operands. Adding operators find the sum and difference of 2 operands.

Table 5-2 Signs of Division/Remainder Division Operation Result

a / b		b	
		+	-
a	+	+	-
	-	-	+

a % b		b	
		+	-
a	+	+	+
	-	-	-

Remark a, b indicates each operand.

Division is performed with 2 integers whose sign, if any, is removed through the usual arithmetic conversion and the result will be truncated towards 0 if necessary. Likewise, a remainder or modulo division operation is performed with 2 integers whose sign, if any, is removed through the usual arithmetic conversion. Table 5-2 shows the results of calculations only on the signs of 2 operands in division and remainder division, respectively.

* operator

SYNTAX

```
E1 * E2
```

FUNCTION

- The binary * (multiplication) operator performs normal multiplication on 2 operands and returns the product.

/ operator

SYNTAX

```
E1 / E2
```

FUNCTION

- The / operator performs normal division on 2 operands and returns the quotient.

% operator

SYNTAX

```
E1 % E2
```

FUNCTION

- The % operator performs a remainder (or modulo division) operation on 2 operands and returns the remainder in the result.

+ operator

SYNTAX

```
E1 + E2
```

FUNCTION

- The + operator performs addition on 2 operands and returns the sum of the 2 numbers.

- operator

SYNTAX

```
E1 - E2
```

FUNCTION

- The - operator performs subtraction on 2 operands and returns the difference between the 2 numbers (the first operand minus the second operand).

5.6 Bitwise Shift Operators

A shift operator shifts its first (left) operand to the direction (left or right) indicated by the operator by the number of bits specified by its second operand.

The types of shift operators are given below.

- << operator
- >> operator

Table 5-3 Shift Operations

a << b		b[Note]
a	+	0
	-	0

a >> b		b[Note]
a	+	0
	-	-1

Note The table indicates when the right operand is greater than the number of bits in the left operand or when an overflow occurs in the result of the shift operation.
If the right operand is negative, the value is processed as an unsigned positive number.

Remark a, b indicates each operand.

<< operator

SYNTAX

```
E1 << E2
```

FUNCTION

- The binary << (left shift) operator shifts the left operand to the left the number of bits specified by the right operand and fills zeros in vacated bits. If the left operand E1 has an unsigned type in "E1 << E2", the result will become a value obtained by multiplying "E1" by the "E2th" power of 2.

>> operator

SYNTAX

```
E1 >> E2
```

FUNCTION

- The binary >> (right shift) operator shifts the left operand to the right the number of bits specified by the right operand.
- If "E1" is unsigned, zeros are filled in vacated bits (Logical shift).
- If "E1" is signed, a copy of the sign bit is filled in vacated bits.
- If "E1" is unsigned or signed and have a non-negative value in "E1>>E2", the result will become a value obtained by dividing "E1" by the "E2th" power of 2.

5.7 Relational Operators

There are 2 types of operators to indicate the relationship between 2 operands: "relational operator" and "equality operator".

The relational operator indicates the value relationship between 2 operands such as greater than and less than. The equality operators indicate that 2 operands are equal or not equal.

The relational operators and equality operators are shown below.

- < operator
- > operator
- <= operator
- >= operator
- == operator
- != operator

The value relationship between 2 pointers compared by relational operators is determined by the relative location in the address space of the object indicated by the pointer.

In the CC78K0R, relational operators and equality operators generate "1" if the specified relationship is true and "0" if it is false. The results have int type.

< operator

SYNTAX

```
E1 < E2
```

FUNCTION

- The < (less than) operator returns "1" if the left operand is less than the right operand; otherwise, "0" is returned.

> operator

SYNTAX

```
E1 > E2
```

FUNCTION

- The > (greater than) operator returns "1" if the left operand is greater than the right operand; otherwise, "0" is returned.

<= operator

SYNTAX

```
E1 <= E2
```

FUNCTION

- The <= (less than or equal) operator returns "1" if the left operand is less than or equal to the right operand; otherwise, "0" is returned.

>= operator

SYNTAX

```
E1 >= E2
```

FUNCTION

- The >= (greater than or equal) operator returns "1" if the left operand is greater than or equal to the right operand; otherwise, "0" is returned.

== operator

SYNTAX

```
E1 == E2
```

FUNCTION

- The == (equal) operator returns "1" if its 2 operands are equal to each other; otherwise, "0" is returned.

!= operator

SYNTAX

```
E1 != E2
```

FUNCTION

- The != (not equal) operator returns "1" if both operands are not equal to each other; otherwise, "0" is returned.

5.8 Bitwise Logical Operators

Bitwise logical operators perform a specified logical operation on the value of an object in bit units.

Each logical operation is indicated by the operators shown below.

- Bitwise AND operator (&)
- Bitwise XOR operator (^)
- Bitwise inclusive OR operator (|)

Bitwise AND operator (&)

SYNTAX

```
E1 & E2
```

FUNCTION

- The binary "&" operator is a bitwise AND operator which returns an integral value that has "1" bits in positions where both operands have "1" bits and that has "0" bits everywhere else.
- The bitwise AND operator must be specified with an "& operator".

		Value of Each Bit in Left Operand	
		1	0
Value of Each Bit in Right Operand	1	1	0
	0	0	0

Bitwise XOR operator (^)

SYNTAX

```
E1 ^ E2
```

FUNCTION

- The binary " ^ " (caret) operator is a bitwise exclusive OR operator which returns an integral value that has a "1" bit in each position where exactly one of the operands has a "1" bit and that has a "0" bit in each position where both operands have a "1" bit or both have a "0" bit.

		Value of Each Bit in Left Operand	
		1	0
Value of Each Bit in Right Operand	1	0	1
	0	1	0

Bitwise inclusive OR operator (|)

SYNTAX

```
E1 | E2
```

FUNCTION

- The binary " | " operator is a bitwise inclusive OR operator which returns an integral value that has a "1" bit in each position where at least one of the operands has a "1" bit and that has a "0" bit in each position where both operands have a "0" bit.

		Value of Each Bit in Left Operand	
		1	0
Value of Each Bit in Right Operand	1	1	1
	0	1	0

5.9 Logical Operators

Logical operators perform logical OR and logical AND operations.

A logical OR operation is specified with a logical OR operator, and a logical AND operation is specified with a logical AND operator.

Each operator is shown below.

- Logical AND operator (&&)
- Logical OR operator (||)

Each operand of both the operators returns the value of int type "0" or "1".

Logical AND operator (&&)

SYNTAX

```
E1 && E2
```

FUNCTION

- The && operator performs logical AND operation on 2 operands and returns a "1" if both operands have nonzero values.

  Otherwise, a "0" is returned. The type of the result is int.

		Value of Left Operand	
		Zero	Nonzero
Value of Right Operand	Zero	0	0
	Nonzero	0	1

NOTE

- This operator always evaluates its operands from left to right. If the value of the left operand is "0", the right operand is not evaluated.

Logical OR operator (||)

SYNTAX

```
E1 || E2
```

FUNCTION

- The | | operator performs logical OR operation on 2 operands and returns a "0" if both operands are zero. Otherwise, a "1" is returned. The type of result is int.

		Value of Left Operand	
		Zero	Nonzero
Value of Right Operand	Zero	0	1
	Nonzero	1	1

NOTE

- This operator always evaluates its operands from left to right. If the value of the left operand is nonzero, the right operand is not evaluated.

5.10 Conditional Operator

Conditional operators judge the processing to be performed next by the value of the first operand. Conditional operators judge by "?" and ":".

The types of conditional operators are given below.

- Conditional operator (? :)

Conditional operator (? :)

SYNTAX

```
1st-operand ? 2nd-operand : 3rd-operand
```

FUNCTION

- If the value of the first operand is nonzero, it evaluates the second operand before the colon. If the value of the first operand is zero, it evaluates the third operand after the colon.

  The result of the entire conditional expression will be the value of the second or third operand.

EXAMPLE

```
#define TRUE    1
#define FALSE   0

char    flag ;
int     ret ;

int     func ( void ) {
        ret = flag ? TRUE : FALSE ;
        return  ret ;
}
```

NOTE

- If both the second and third operand types are arithmetic types, normal arithmetic type conversion is performed to make them common types. The type of result is the common type.

  If both the operand types are structure types or union types, the result becomes those types. If both the operand types are void types, the result is void type.

5.11 Assignment Operators

Assignment operators include a simple assignment expression that stores the right operand in the left operand and a compound assignment expression that stores the result of an operation on both operands in the left operand.

The assignment operators are shown below.

- Simple assignment operator (=)
- Compound assignment operators (*= /= %= += -= <<= >>= &= ^= |=)

Simple assignment operator (=)

SYNTAX

```
E1 = E2
```

FUNCTION

- The = (simple assignment) operator converts the right operand (expression) to the type of the left operand (Lvalue) before the value is stored.

  In the following example, the value of an int type to be returned from the function by the type conversion of the simple assignment expression will be converted to a char type and an overflow in the result will be truncated. And the comparison of the value with "-1" will be made after the value is converted back to the int type. If the variable "c" declared without qualifier is not interpreted as unsigned char, the result of the variable will not become negative and its comparison with "-1" will never result in equal. In such a case, the variable "c" must be declared with an int type to ensure complete portability.

```
int     f ( void ) ;
char    c ;

if ( ( c = f ( ) ) == -1 ) {
    :
} else {
    :
}
```

Compound assignment operators (*= /= %= += -= <<= >>= &= ^= |=)

SYNTAX

```
E1 *= E2
E1 /= E2
E1 %= E2
E1 += E2
E1 -= E2
E1 <<= E2
E1 >>= E2
E1 &= E2
E1 ^= E2
E1 |= E2
```

FUNCTION

- The compound assignment operators perform a specified operation on both operands and stores the result in the left operand. The value to be stored in the left operand will be converted to the type of Lvalue (left operand).
- The compound assignment expression "E1 op = E2" (where op indicates a suitable binary operator) is equivalent to the simple assignment expression "E1 = E1 op (E2)", except that the Lvalue (E1) is only evaluated once. The following compound assignment expressions will produce the same result as the respective simple assignment expressions on the right.

```
a *= b ;        a = a * b ;
a /= b ;        a = a / b ;
a %= b ;        a = a % b ;
a += b ;        a = a + b ;
a -= b ;        a = a - b ;
a <<= b ;       a = a << b ;
a >>= b ;       a = a >> b ;
a &= b ;        a = a & b ;
a ^= b ;        a = a ^ b ;
a |= b ;        a = a | b ;
```

5.12 Comma Operator

The types of comma operators are given below.

- Comma operator (,)

Comma operator (,)

SYNTAX

```
E1 , E2
```

FUNCTION

- The comma operator evaluates the left operand as a void type (that is, ignores its value) and then evaluates the right operand.

  The type and value of the result of the comma expression are the type and value of the right operand.

- In contents where a comma has another meaning (as in a list of function arguments or in a list of variable initializations), comma expressions must be enclosed in parentheses. In other words, the comma operator described in this chapter will not appear in such a list.

- In the following example, the comma operator finds the value of the second argument of the function "f ( )". The value of the second argument becomes 5.

```
int     a , c , t ;

void    main ( void )  {
        f ( a , ( t = 3 , t + 2 ) , c ) ;
}
```

5.13 Constant Expressions

Constant expressions include general integral constant expressions, arithmetic constant expressions, address constant expressions, and initialization constant expressions.

Most of these constant expressions can be calculated at translation time instead of execution time.

In a constant expression, the following operators cannot be used except when they appear inside sizeof expressions:

- Assignment operators
- Increment operators
- Decrement operators
- Function call operator
- Comma operator

(1) General integral constant expression

A general integral constant expression has a general integral type.

The following operands may be used:

- Integer constants
- Enumerated value constants
- Character constants
- sizeof expressions
- Floating point constants

(2) Arithmetic constant expression

An arithmetic constant expression has an integral type.

The following operands may be used:

- Integer constants
- Enumerated value constants
- Character constants
- sizeof expressions
- Floating point constants

(3) Address constant expression

An address constant expression is a pointer to an object that has a static storage duration or a pointer to a function locator.

Such an expression must be created explicitly using the unary & operator or implicitly using an expression with an array type or function type.

The following operands may be used. However, none of these operators can be used to access the value of an object.

- Array subscript operator "[ ]"
- "." (dot) operator
- "->" (arrow) operator
- "&" address operator
- " * " indirection operator
- Pointer casts

CHAPTER 6 CONTROL STRUCTURES OF C LANGUAGE

This chapter describes the program control structures of C language and the statements to be executed in C.

Generally speaking, no matter how a process is complicated, it can be expressed with 3 basic control structures. These 3 control structures are: Sequential, Conditional control (Selection), and Iteration. Branch is used to change the flow of a program by force.

(1) Sequential processing

Statements in a program are executed one by one from top to bottom in the order of their description in the program.

(2) Conditional control (selection) processing

According to the status of the program under execution, the next executable statement is selected and executed.

The selection condition is specified in a control statement. The control statement determines which of the 2 alternative statement groups or multiway (three or more) alternative statement groups is to be executed.

(3) Looping (iteration) processing

The same processing is executed two or more times.

The execution of an executable statement is repeated a specified number of times during the condition indicated by the control statement.

(4) Branch processing

C is caused to exit from the current program flow and control is transferred to a specified label.

Program execution starts from the statement next to the specified label.

There are 6 types of statements used in C.

- Labeled Statements

  Causes branch according to the value of switch statement and the destination of goto statement

- Compound Statements or Blocks

  Collects two or more statements to be processed as 1 unit

- Expression Statements and Null Statements

  A statement consisting of an expression and a semicolon

- Conditional Control Statements

  Selects a statement out of several statements according to the value of the expression

- Looping Statements

  Repeatedly performs a statement called the body of a loop until the control expression becomes equal to 0.

- Branch Statements

  Causes unconditional branch to different destination

Description example of these statements is shown below.

```
#define SIZE    10
#define TRUE    1
#define FALSE   0

extern  void    putchar ( char ) ;
extern  void    lprintf ( char * , int ) ;

charmark [ SIZE + 1 ] ;

void    main ( void ) {
    int     i , prime , k , count ;

    count = 0 ;
    for ( i = 0 ; i <= SIZE ; i++ )     /* for :    Looping statement */
        mark [ i ] = TRUE ;
    for ( i = 0 ; i <= SIZE ; i++ )  {  /* for :    Looping statement */
        if ( mark [ i ] )  {            /* if :     Conditional statement */
            prime = i + i + 3 ;
            lprintf ( "%d " , prime ) ;
            if ( ( count%8 ) == 0 )     /* if :     Conditional statement */
                putchar ( '\n' ) ;
            for ( k = i + prime ; k <= SIZE ; k += prime )
                mark [ k ] = FALSE ;
        }
    }
    lprintf ( "Total %d\n" , count ) ;

loop1: ;                                /* loop1 :  Labeled statement */
    goto    loop1 ;                     /* goto :   Branch statement */
}
```

6.1 Labeled Statements

A labeled statement specifies the destination of switch or goto statement.

The switch statement selects the statement specified by a control expression from among statements with two or more options. The labeled statement becomes the label of the statement to be executed by the switch statement.

The goto statement causes unconditional branching to the applicable label from the normal flow of processing.

The types of labeled statements are given below.

- case label
- default label

case label

SYNTAX

```
case constant-expression : statement
```

FUNCTION

- case labels are used only in the body of a switch statement to enumerate values to be taken by the control expression of the switch statement.

EXAMPLE 1

```
int     f ( void ) , i ;

switch ( f ( ) ) {
        case 1 :
                i = i + 4 ;
                break ;
        case 2 :
                i = i + 3 ;
                break ;
        case 3 :
                i = i + 2 ;
}
```

- In EXAMPLE 1, if the return value of f( ) is 1, the first case clause (statement) is selected and the expression "i=i+4" is executed. Likewise, if the return value of f( ) is 2 or 3, the second or third case statement is selected, respectively. Each break statement in the above example is to break out of the switch body.

  As in this example, case labels are used when two or more options are involved.

EXAMPLE 2

```
int     i ;

i = 2 ;
switch ( i ) {
        case 1 :
                i = i + 4 ;
        case 2 :
                i = i + 3 ;
        case 3 :
                i = i + 2 ;
}
```

EXPLANATION

- In example 2, the processing starts in the second case statement since i is 2. The third statement is also consecutively performed since the case statement does not include a break statement.
  Thus, if the constant expression and the control expression in the case statement match, the programs thereafter are performed sequentially. A break statement is used to exit the switch statement.

default label

SYNTAX

```
default : statement
```

FUNCTION

- A default label is a special case label used only in the body of a switch statement to specify a process to be executed by C if the value of the control expression does not match any of the case constants.

EXAMPLE

```
int     f ( void ) ,  i ;

switch ( f ( ) ) {
        case 1 :
                i = i + 4 ;
                break ;
        case 2 :
                i = i + 3 ;
                break ;
        case 3 :
                i = i + 2 ;
        default :
                i = 1 ;
}
```

- In the above example, if the return value of f( ) is 1, 2, or 3, the corresponding case clause (statement) is selected and the expression that follows the case label is executed. Each break statement in the above example is to break out of the switch body. If the return value of f( ) is other than 1 to 3, the expression that follows the default label is executed. In this case, the value of i becomes 1.

6.2 Compound Statements or Blocks

A compound statement or block is synonymous to each other and consists of two or more statements grouped together with enclosing braces and executed as 1 unit syntax-wise.

In other words, by enclosing zero or more declarations followed by zero or more statements all in braces, these statements can be processed as a compound statement whenever a single statement is expected.

6.3 Expression Statements and Null Statements

An expression statement consists of a statement and a semicolon. A null statement consists of only a semicolon and is used for labels that require a statement and in looping that do not need any body.

The description examples of expression statements and null statements are given below.

As in the following example, for a function to be called as an expression statement merely to obtain side effects, the value of its return value can be discarded by using a cast expression.

```
int     p ( int ) ;

( void ) p ( 0 ) ;
```

A null statement can be used as the body of a looping statement as shown below.

```
char    *s ;

while ( *s++ != ' 0 ' ) ;
```

In addition, it can be used to place a label before a brace "}" which closes a compound statement as shown below.

```
while ( loop1 )  {
          :
        while ( loop2 )  {
                  :
                if ( want_out )
                        goto    end_loop1 ;
                  :
        }
end_loop1 : ;
}
```

6.4 Conditional Control Statements

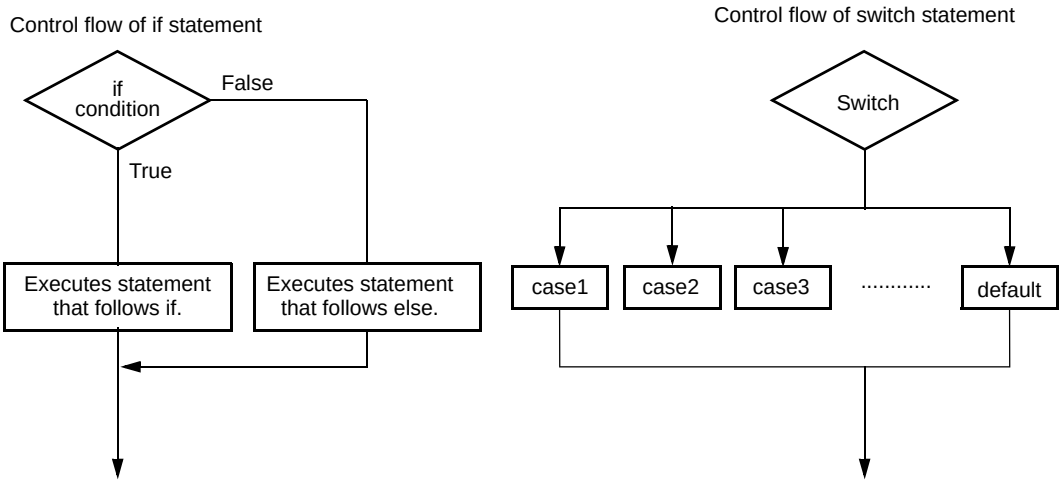
Conditional control (or selection) statements include if and switch statements.

The if or switch statement allows the program to choose one of several groups of statements to execute, based on the value of the control expression enclosed in parentheses. The types of conditional control statements are given below.

- if and if ... else statements
- switch statement

The control flows of if and switch statements are illustrated in Figure 6-1 below.

Figure 6-1 Control Flows of Conditional Control Statements



if and if ... else statements

SYNTAX

```
if ( expression ) statement
if ( expression ) statement-1 else statement-2
```

FUNCTION

- An if statement executes the statement that follows the control expression enclosed in parentheses if the value of the control expression is nonzero.
- An if ... else statement executes the statement-1 that follows the control expression if the value of the control expression is nonzero or the statement-2 that follows else if the value of the control expression is zero.

EXAMPLE

```
unsigned char   uc ;

if ( uc < 10 ) {
        /* 111 */
} else if ( uc < 20 ) {
        /* 222 */
} else {
        /* 333 */
}
```

- In the above example, if the value of uc is less than 10 based on the control expression in the if statement, the block "{/* 111 */}" is executed. If the value is greater than 10, the block "{/* 222 */}" is executed.

NOTE

- When the processing after if statement/if...else statement is not enclosed with "{ }", only the processing of a line after the if statement/if...else statement is performed regarding it as the body.

switch statement

SYNTAX

```
switch ( expression ) statement
```

FUNCTION

- A switch statement has a multiway branching structure and passes control to one of a series of statements that have the case labels in the switch body depending on the value of the control expression enclosed in parentheses.

  If no case label that corresponds to the control expression exists, the statement that follows the default label is executed. If no default label exists, no statement is executed.

EXAMPLE

```
extern void     func ( void ) ;
extern void     error_mode ( void ) ;

unsigned char   mode ;

switch ( mode ) {
        case 2 :
                mode = 8 ;
                break ;
        case 4 :
                mode = 2 ;
                break ;
        case 8 :
                func ( ) ;
        default :
                error_mode ( ) ;
}
```

NOTE

- The same value cannot be set in each case label in the switch body. Only 1 default label can be used in the switch body.

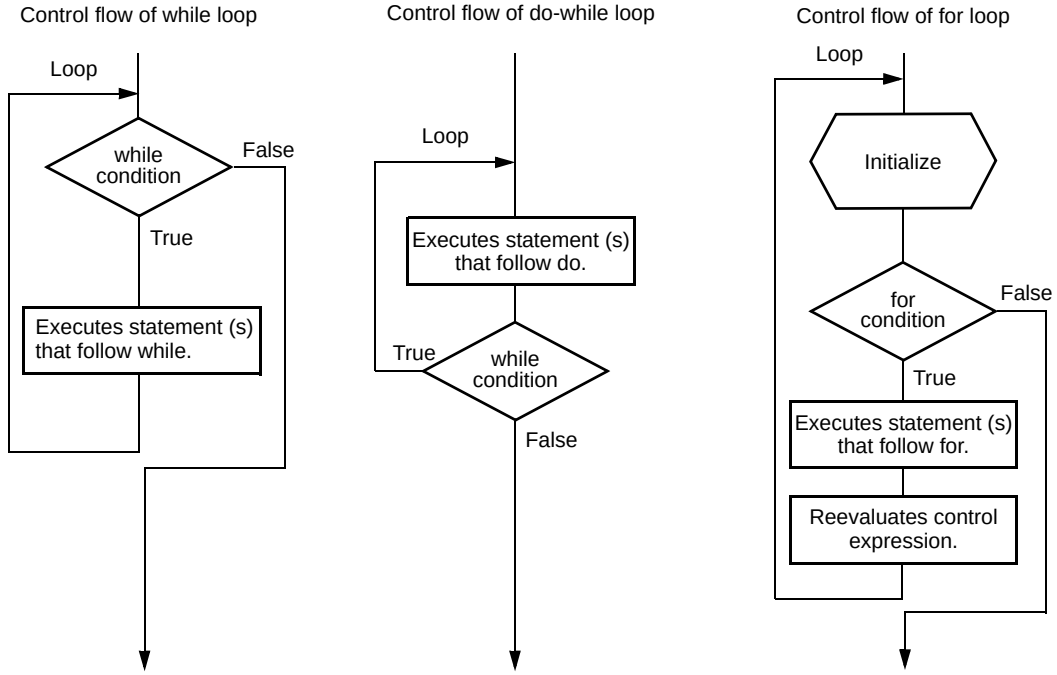
6.5 Looping Statements

A looping (or iteration) statement executes a group of statements in the loop body as long as the value of the control expression enclosed in parentheses is True (nonzero). C has the following 3 types of looping statements:

- while statement
- do statement
- for statement

The control flow of each type of looping statement is illustrated in Figure 6-2 below.

Figure 6-2 Control Flows of Looping Statements



while statement

SYNTAX

```
while ( expression ) statement
```

FUNCTION

- A while statement executes one or more statements (the body of the while loop) several times as long as the value of the control expression enclosed in parentheses is True (nonzero).
  The while statement evaluates the control expression before executing its loop body.

EXAMPLE

```
int     i , x ;

i = 1 ,  x = 0 ;

while ( i < 11 )  {
        x += i ;
        i++ ;
}
```

- The above example finds the sum total of integers from 1 to 10 for x. The 2 statements enclosed in brace brackets are the body of this while loop. The control expression "i<11" returns 0 if the value of i becomes 11. For this reason, the loop body is executed repeatedly as long as the value of i is less than 11 (between 1 and 10).
- "while(1) {statement}" is used to endlessly perform a loop statement.

do statement

SYNTAX

```
do statements while ( expression ) ;
```

FUNCTION

- A do statement executes the body of the loop and then tests the control expression enclosed in parentheses to see if its value is True (nonzero).

  The do statement evaluates the control expression after the loop body has been executed.

EXAMPLE

```
int     i , x ;

i = 1 ,  x = 0 ;

do  {
        x += i ;
        i++ ;
} while ( i < 11 ) ;
```

- The above example finds the sum total of integers from 1 to 10 for x. The 2 statements enclosed in brace brackets are the body of this do ... while loop. The control expression "i<11" returns 0 if the value of i becomes 11. For this reason, the loop body is executed repeatedly as long as the value of i is less than 11 (between 1 and 10). The body of the loop is always performed once or more since the control expression of a do statement is evaluated after execution.

for statement

SYNTAX

```
for ( 1st-expression ; 2nd-expression ; 3rd-expression ) statements
```

FUNCTION

- A for statement executes the body of the for loop a specified number of times as long as the value of the control expression is nonzero (True).

  Of the 3 expressions inside the parentheses separated by semicolons, the first expression is an initializing statement to initialize a variable to be used as a counter and execute only once in the beginning of the loop, the second is the control expression for testing the counter value, and the third is a step statement executed in the end of every loop and reevaluate the variable after the execution.

EXAMPLE

```
int     i , x = 0 ;

for ( i = 1 ; i < 11 ; ++i )
        x += i ;
```

- The above example finds the sum total of integers from 1 to 10 for x. "x+=i" is the body of this for loop. The control expression "i<11" returns 0 if the value of i becomes 11. For this reason, the loop body is executed repeatedly as long as the value of i is less than 11 (between 1 and 10).

NOTE

- When the processing after for statement is not enclosed with "{ }", only the processing of a line after the for statements is regarded as the body of the loop of the for statement.
- The first and the third expression of a for statement can be omitted. When the second expression is omitted, it is replaced with a constant other than 0. The description of "for (; ;) statement" is used to endlessly perform the body of the loop.

6.6 Branch Statements

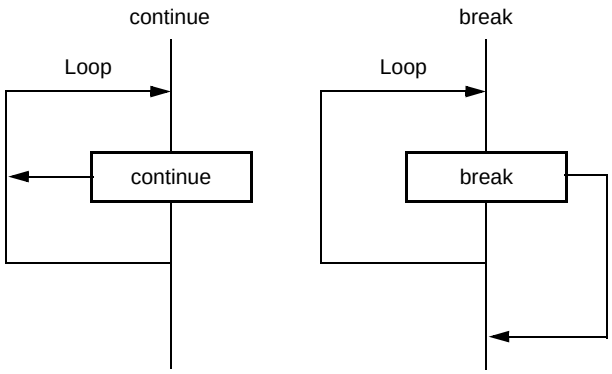
A branch statement is used to exit from the current control flow and transfer control to elsewhere in the program.

Branch statements include the following 4 statements:

- goto statement
- continue statement
- break statement
- return statement

The control flow of each type of branch statement is shown in Figure 6-3.

Figure 6-3 Control Flows of Branch Statements



goto statement

SYNTAX

```
goto identifier ;
```

FUNCTION

- A goto statement causes program execution to unconditionally jump to the label name specified in the goto statement within the current function.

EXAMPLE

```
do {
          :
        goto    point ;
          :
} while ( i < 11 ) ;
          :
point : ;
```

- In the above example, when control is passed to the goto statement, C jumps out of the current do ... while loop processing without condition and transfers control to the statement next to "point".

NOTE

- The label name (branch destination) to be specified in a goto statement must have been specified within the current function that includes the goto statement. In other words, a goto can branch only within the current function - not from one function to another.

continue statement

SYNTAX

```
continue ;
```

FUNCTION

- A continue statement is used in the body of loops in a looping statement. continue ends one cycle of the loop by transferring control to the end of the loop body. When a continue statement is enclosed by more than 1 loop, it ends a cycle of the smallest enclosing loop.

EXAMPLE

```
while ( i < 11 ) {
          :
        continue ;
          :
contin : ;
}
```

- In the above example, when the while loop processing by C reaches the continue statement, C unconditionally branches to the label "contin". The label "contin" indicates the branch destination and may be omitted. The same branching operation may be performed by using "goto contin;" instead of continue.

NOTE

- A continue statement can only be used as the body of a loop or in the body of loops.

break statement

SYNTAX

```
break ;
```

FUNCTION

- A break statement may appear in the body of a loop and in the body of a switch statement and causes control to be transferred to the statement next to the loop or switch statement.

EXAMPLE

```
int             i ;
unsigned char   count  , flag ;

  :
for ( i = 0 ; i < 20 ; i++ )  {
        switch ( count )  {
                case 10 :
                        flag = 1 ;
                        break ;         /* Exit switch statement */
                default :
                        func ( ) ;
        }
        if ( flag )
                break ;                 /* Exit for loop */
}
```

- In the above example, break statements are used so that more than required evaluations are not performed in the body of the switch statement. If the corresponding case label is found in evaluating the switch statement, the break statement causes C to exit from the switch statement.

NOTE

- A break statement can only be used as the body of a looping or switch statement or in the loop or switch body.

return statement

SYNTAX

```
return expression ;
```

FUNCTION

- A return statement exits the function that includes the return and passes controls to the function that called the return, and it calls and returns the value of the return statement expression as the value of the function call expression.
- Two or more return statements may be used in a function.
- Using the closing brace bracket "} " at the end of a function produces the same result as when a return statement without expression is executed.

EXAMPLE

```
int     f ( int ) ;

void    main ( void )  {
        int     i = 0 ,  y = 0 ;
          :

        y = f ( i ) ;
          :
}

int     f ( int  i ) {
        int x = 0 ;
          :
        return ( x ) ;
}
```

- In the above example, when control is passed to the return statement, the function f( ) returns a value to the function main. Because the value of the variable "x" is returned as the return value, the assignment operator causes the variable "y" to be substituted with the value of the variable "x".

NOTE

- With a void type function, an expression that indicates a return value cannot be used for a return statement.

CHAPTER 7 STRUCTURES AND UNIONS

A structure or union is a collection of member objects that have different types and grouped under 1 given name.

The member objects of a structure are allocated successively to memory space, while the member objects of a union share the same memory.

7.1 Structures

As mentioned earlier, a structure is a collection of member objects successively allocated to memory space.

(1) Declaration of structure and structure variable

A structure declaration list and a structure variable are declared with the keyword "struct".

Any name called a tag name can be given to the structure declaration list.

Subsequently, the structure variables of the same structure may be declared using this tag name.

[Declaration of structure]

```
struct  tag-name {
        structure-declaration-list
} variable-name ;
```

In the following example, in the first struct declaration, int type array "code", char type arrays name, addr, and tel which have a tag name "data" are specified and no1 is declared as the structure variable. In the second struct declaration, the structure variables no2, no3, no4, and no5 that are of the same structure as no1 are declared.

```
struct  data {
        int     code ;
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no1 ;
struct  data    no2 , no3 , no4 , no5 ;
```

(2) Structure declaration list

A structure declaration list specifies the structure of a structure type to be declared.

Individual elements in the structure declaration list are called members and an area is allocated for each of these members in the order of their declaration. In the following **[Example of structure declaration list]**, an area is allocated in the order of variable a, array b, and 2-dimensional array c.

Neither an incomplete type (an array of unknown size) nor a function type can be specified as the type of each member. Therefore, the structure itself cannot be included in the structure declaration list.

Each member can have any object type other than the above 2 types. A bit field which specifies each member in bits can also be specified.

If a variable takes a binary value "0" or "1", the minimum required of bits is specified as 1 for a bit field. By this specification of the minimum required number of bits with the bit field, two or more members can be stored in an integer area.

[Example of structure declaration list]

```
int     a ;
char    b [ 7 ] ;
char    c [ 5 ] [ 10 ] ;
```

[Example of bit field declaration]

```
struct  bf_tag {
        unsigned int    a : 2 ;
        unsigned int    b : 3 ;     bit field
        unsigned int    c : 1 ;
} bit_field ;
```

(3) Arrays and pointers

Structure variables may also be declared as an array or referenced using a pointer.

In structure arrays, the elements of arrays are also handled as structures.

[Structure arrays]

An array of structures is declared in the same ways as other objects.

```
struct  data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} ;
struct  data    no [ 5 ] ;
```

[Structure pointers]

A pointer to a structure has the characteristics of the structure indicated by the pointer. In other words, if a structure pointer is incremented, adding the size of the structure to the pointer points to the next structure.

In the following example, "dt_p" is a pointer to the value of "struct data" type. Here, if the pointer "dt_p" is incremented, the pointer becomes the same value as "&no [1]".

```
struct  data    no [ 5 ] ;
struct  data    *dt_p = no ;
```

(4) How to refer to structure members

A structure member may be referenced in 2 ways: one by using a structure variable and the other by using a pointer to a variable.

[Reference by using a structure variable]

The "." (dot) operator is used for referring to a structure member by using a structure variable.

```
struct  data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no [ 5 ] = { "NAME" , "ADDR" , "TEL" } , *data_ptr = no ;

void    main ( void ) {
        char    c ;

        c = no [ 0 ] . name [ 1 ] ;
}
```

[Reference by using a pointer to a variable]

The "->" (arrow) operator is used for referring to a structure member by using a pointer to a variable.

```
struct  data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no [ 5 ] = { "NAME" , "ADDR" , "TEL" } , *data_ptr = no ;

void    main ( void ) {
        char    c ;

        data_ptr -> tel [ 3 ] = 'E' ;
}
```

7.2 Unions

As mentioned earlier, a union is a collection of members which share the same memory space (or overlap in memory).

(1) Declaration of union and union variable

A union declaration list and a union variable are declared with the keyword "union". Any name called a tag name can be given to the union declaration list. Subsequently, the union variables of the same union may be declared using this tag name.

[Declaration of union]

```
union tag-name { union-declaration-list } variable-name ;
```

In the following example, in the first union declaration, char type arrays "name", "addr", and "tel" which have a tag name "data" are specified and "no1" is declared as the union variable. In the second union declaration, the union variables "no2, no3, no4, and no5" which are of the same union as "no1" are declared.

```
union   data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no1 ;
union   data    no2 , no3 , no4 , no5 ;
```

(2) Union declaration list

A union declaration list specifies the structure of a union type to be declared.

Each element on the union declaration list is called a member. Declared members are allocated to the same area. In the following **[Example of union declaration list]**, an area is allocated to "c", which becomes the largest area of the members. The other members are not allocated new areas but use the same area.

Neither an incomplete type (an array of unknown size) nor a function type can be specified as the type of each member same as the union declaration list.

Each member can have any object type other than the above 2 types.

[Union declaration list]

```
int     a ;
char    b [ 7 ] ;
char    c [ 5 ] [ 10 ] ;
```

(3) Union arrays and pointers

Union variables may also be declared as an array or referenced using a pointer (in much the same way as structure arrays and pointers).

[Union arrays]

An array of unions is declared in the same ways as other objects.

```
union   data {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} ;
union   data    no [ 5 ] ;
```

[Union pointers]

A pointer to a union has the characteristics of the union indicated by the pointer. In other words, if a union pointer is incremented, adding the size of the union to the pointer points to the next union.

In the following example, "dt_p" is a pointer to the value of "union data" type.

```
union   data    no [ 5 ] ;
union   data    *dt_p = no ;
```

(4) How to refer to union members

A union member (or union element) may be referenced in 2 ways: one by using a union variable and the other by using a pointer to a variable.

[Reference by using a union variable]

The "." (dot) operator is used for referring to a union member by using a union variable.

```
union   data {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no [ 5 ] = { "NAME" , "ADDR" , "TEL" } ;

void    main ( void ) {
        no [ 0 ] .addr [ 10 ] = 'B' ;
            :
}
```

[Reference by using a pointer to a variable]

The "->" (arrow) operator is used for referring to a union member by using a pointer to a variable.

```
union   data {
        char        name [ 12 ] ;
        char        addr [ 50 ] ;
        char        tel [ 12 ] ;
} *data_ptr ;

void    main ( void ) {
        data_ptr -> name [ 1 ] = 'N' ;
            :
}
```

CHAPTER 8 EXTERNAL DEFINITIONS

In a program, lists of external declaration come after the preprocessing. These declaration are referred to as "external declaration" because they appear outside a function and have effective file ranges.

A declaration to give a name to external objects by identifiers or a declaration to secure storage for a function is called an external definition. If an identifier declared with external linkage is used in an expression (except the operand part of the sizeof operator), only 1 external definition for the identifier must exist somewhere in the entire program.

The syntax of external definitions is given below.

```
#define TRUE    1
#define FALSE   0
#define SIZE    200
void    printf ( char * , int ) ;
void    putchar ( char c ) ;

char    mark [ SIZE + 1 ] ;     /* External object declaration */

void    main ( void ) {
        int     i , prime , k , count ;

        count = 0 ;

        for ( i = 0 ; i <= SIZE ; i++ )
                mark [ i ] = TRUE ;
        for ( i = 0 ; i <= SIZE ; i++ ) {
                if ( mark [ i ] ) {
                        prime = i + i + 3 ;
                        printf ( "%d" , prime ) ;
                        count++ ;
                        if ( ( count%8 ) == 0 ) putchar ( '\n' ) ;
                        for ( k = i + prime ; k <= SIZE ; k += prime )
                                mark [ k ] = FALSE ;
                }
        }
        printf ( "Total %d\n" , count ) ;

loop1 :
        goto    loop1 ;
}
```

8.1 Function Definition

A function definition is an external definition that begins with a declaration of the function.

If the storage class specifier is omitted from the declaration, "extern" is assumed to have been defined. An external function definition means that the defined function may be referenced from other files. For example, in a program consisting of two or more files, if a function in another file is to be referenced, this function must be defined externally.

The storage class specifier of an external function is extern or static. If a function is declared as extern, the function can be referenced from another file. If declared as static, it cannot be referenced from another file.

In the following example, the storage class specifier is "extern" and the type specifier is "int". These two are default values and thus may be omitted from specification. The function declarator is "max(int a, int b)" and the body of the function is "{return a > b ? a : b ;)".

[Example of function definition]

```
extern  int     max ( int a , int b ){
        return a > b ? a : b ;
}
```

Because this function definition specifies a parameter type in the function declaration, the type of argument is forcedly converted by the compiler. By using the form of an identifier list for the parameters, this type conversion can be described. An example of this identifier list is shown below.

```
extern  int     max ( a , b )
int     a , b ;
{
       return a > b ? a : b ;
}
```

As an argument to a function call, the address of the function may be passed. By using the function name in the expression, a pointer to the function can be generated.

```
int     f ( void ) ;

void    main ( void )  {
          :
        g ( f ) ;
          :
}
```

In the above example, the function g is passed to the function f by a pointer that points to the function f. The function g must be defined in either of the following 2 ways:

```
void    g ( int ( *funcp ) ( void ) ){
        ( *funcp ) ( ) ;        /* Or funcp ( ) ; */
}
```

or

```
void    g ( int func ( void ) ){
        func ( ) ;              /* Or ( *func ) ( ) ; */
}
```

8.2 External Object Definitions

An external object definition refers to the declaration of an identifier for an object that has file scope or initializer. If the declaration of an identifier for an object which has file scope has no initializer without storage class specification or has storage class static, the object definition is considered to be temporary, because it becomes a declaration which has file scope with initializer 0.

Examples of external object definitions are shown in the table below.

Table 8-1 Example of External Object Definition

```
int             i1 = 1 ;    /* Definition with external linkage */
static  int     i2 = 2 ;    /* Definition with internal linkage */
extern  int     i3 = 3 ;    /* Definition with external linkage */
int             i4 ;        /* Temporary definition with external linkage */
static  int     i5 ;        /* Temporary definition with internal linkage */

int     i1 ;    /* Valid temporary definition which refers to previous declaration */
int     i2 ;    /* Violation of linkage rule */
int     i3 ;    /* Valid temporary definition which refers to previous declaration */
int     i4 ;    /* Valid temporary definition which refers to previous declaration */
int     i5 ;    /* Violation of linkage rule */

extern  int   i1 ;  /* Reference to previous declaration which has external linkage */
extern  int   i2 ;  /* Reference to previous declaration which has internal linkage */
extern  int   i3 ;  /* Reference to previous declaration which has external linkage */
extern  int   i4 ;  /* Reference to previous declaration which has external linkage */
extern  int   i5 ;  /* Reference to previous declaration which has internal linkage */
```

CHAPTER 9 PREPROCESSOR DIRECTIVES (COMPILER DIRECTIVES)

A preprocessor directive is a string of preprocessor tokens between the "#" preprocessor token and the line feed character.

Blank characters that can be used between preprocessor token strings are only spaces and horizontal tabs.

A preprocessor directive specifies the processing performed before compiling a source file. Preprocessor directives include such operations as processing or skipping a part of a source file depending on the condition, obtaining additional code from other source files, and replacing the original source code with other text as in macro expansion.

The followings explain each preprocessor directive.

9.1 Conditional Compilation Directives

Conditional compilation skips part of a source file according to the value of a constant expression.

If the value of the constant expression specified by a conditional compilation directive is 0, the statements that follow the directive are not compiled. The sizeof operator, cast operator, or an enumerated type constant cannot be used in the constant expression of any conditional compilation directive.

The types of Conditional compilation directives are given below.

- #if directive
- #elif directive
- #ifdef directive
- #ifndef directive
- #else directive
- #endif directive

In preprocessor directives, the following unary expressions called defined expressions may be used:

```
defined identifier
or
defined (identifier)
```

The unary expressions return 1 if the identifier has been defined with the #define preprocessor directive and 0 if the identifier has never been defined or its definition has been canceled.

[Example]

In this example, the unary expression returns 1 and compile between #if and #endif because SYM has been defined (for the explanation of #if through #endif, refer to the explanation in the following page and thereafter).

```
#define SYM     0

#if defined     SYM
    :
#endif
```

#if directive

SYNTAX

```
#if constant-expression new-line group
```

FUNCTION

- The #if directive tells the translation phase of C to skip (discard) a section of source code if the value of the constant expression is 0.

EXAMPLE

```
#if FLAG == 0
    :
#endif
```

- In the above example, the constant expression "FLAG == 0" is evaluated to determine whether a set of statements (i.e., source code) between #if and #endif is to be used in the translation phase. If the value of "FLAG" is nonzero, the source code between #if and #endif will be discarded. If the value is zero, the source code between #if and #endif will be translated.

#elif directive

SYNTAX

```
#elif constant-expression new-line group
```

FUNCTION

- The #elif directive normally follows the #if directive. If the value of the constant expression of the #if directive is 0, the constant expression of the #elif directive is evaluated. If the constant expression of the #elif directive is 0, the translation phase of C will skip (discard) the statements (a section of source code) between #elif and #endif.

EXAMPLE

```
#if FLAG == 0
    :
#elif FLAG != 0
    :
#endif
```

- In the above example, the constant expression "FLAG = = 0" or "FLAG! = 0" is evaluated to determine whether a set of statements that follow #if and another set of statements that follow #elif is to be used in the translation phase. If the value of "FLAG" is zero, the source code between #if and #elif will be translated. If the value is nonzero, the source code between #elif and #endif will be translated.

#ifdef directive

SYNTAX

```
#ifdef identifier new-line group
```

FUNCTION

- The #ifdef directive is equivalent to #if defined (identifier)
- If the identifier has been defined with the #define directive, the statements between #ifdef and #endif will be translated. If the identifier has never been defined or its definition has been canceled, the translation phase will skip the source code between #ifdef and #endif.

EXAMPLE

```
#define ON
#ifdef  ON
    :
#endif
```

- In the above example, the identifier "ON" has been defined with #define directive. Thus, the source code between #ifdef and #endif will be translated. If the identifier "ON" has never been defined, the source code between #ifdef and #endif will be discarded.

#ifndef directive

SYNTAX

```
#ifndef identifier new-line group
```

FUNCTION

- The #ifndef directive is equivalent to #if !defined (identifier). If the identifier has never been defined with the #define directive, the source code between #ifndef and #endif will not be translated.

EXAMPLE

```
#define ON

#ifndef ON
    :
#endif
```

- In the above example, the identifier "ON" has been defined with #define directive. Thus, the program between #ifndef and #endif will not be compiled. If the identifier "ON" has never been defined, the program between #ifndef and #endif will be compiled.

#else directive

SYNTAX

```
#else new-line group
```

FUNCTION

- The #else directive tells the translation phase of C to discard a section of source code that follows #else if the identifier of the preceding conditional translation directive is nonzero. The #if, #elif, #ifdef, or #ifndef directive may precede the #else directive.

EXAMPLE

```
#define ON

#ifdef  ON
    :
#else
    :
#endif
```

- In the above example, the identifier "ON" has been defined with #define directive. Thus, the source code between #ifndef and #endif will be translated. If the identifier "ON" has never been defined, the source code between #else and #endif will be translated.

#endif directive

SYNTAX

```
#endif new-line
```

FUNCTION

- The #endif directive indicates the end of a #ifdef block.

EXAMPLE

```
#define ON

#ifdef  ON
    :
#endif
```

- In the above example, "#endif" indicates the end of the #ifdef block (effective range of #ifdef directive).

9.2 Source File Inclusion Directive

The preprocessor directive #include searches for a specified header file and replaces the #include by the entire contents of the specified file.

The #include directive may take one of the following 3 forms for inclusion of other source files:

- #include < > directive
- #include " " directive
- #include preprocessing token string directive

A #include directive may appear in the source obtained by #include. In the CC78K0R, however, there are restrictions for #include directive nest. For the restrictions, refer to Table 1-1.

Remark Preprocessor token string: character string defined by #define directive

#include < > directive

SYNTAX

```
#include        <filename>    new-line
```

FUNCTION

- If the directive form is #include < >, the CC78K0R searches for the header file specified in angle brackets, in the folder specified with compiler option -i, folder specified by the INC78K0R environment variable, and then folder "..\inc78k0r" (for the path through which the CC78K0R is started), and replaces the #include directive line with the entire contents of the specified file.

EXAMPLE

```
#include        <stdio.h>
```

- In the above example, the CC78K0R searches for the file "stdio.h" in the folder specified by the INC78K0R environment variable and folder "..\inc78k0r" (for the path through which the CC78K0R is started), and replaces the directive line "#include <stdio.h>" with the entire contents of the specified file "stdio.h".

#include " " directive

SYNTAX

```
#include        "filename"    new-line
```

FUNCTION

- If the directive form is #include " ", the CC78K0R searches the current folder first for the source file specified in double quotes. If it is not found, the CC78K0R searches the folder specified with compiler option -i, folder specified by the INC78K0R environment variable, and then folder "..\inc78k0r" (for the path through which the CC78K0R is started). When the specified file is found, the CC78K0R then replaces the #include directive line with the entire contents of the file.

EXAMPLE

```
#include        "myprog.h"
```

- In the above example, the CC78K0R searches for the file "myprog.h" in the current folder, the folder specified by the INC78K0R environment variable and folder "..\inc78k0r" (for the path through which the CC78K0R is started), and replaces the directive line #include "myprog.h" with the entire contents of the specified file "myprog.h".

#include preprocessing token string directive

SYNTAX

```
#include        preprocessing-token-string      new-line
```

FUNCTION

- If the directive form is #include preprocessing token string, the header file to be searched is specified by macro replacement and the #include directive line is replaced by the entire contents of the specified file.

EXAMPLE

```
#define         INCFILE "myprog.h"

#include        INCFILE
```

- As a result of the inclusion of other source files with the directive form "#include preprocessing token string new-line", the specified token string must be replaced with <*filename*> or "*filename*" by macro replacement. If the token string is replaced with <*filename*>, the CC78K0R searches for the specified file in the folder specified with compiler option -i, folder specified by the INC78K0R environment variable, and then folder "..\inc78k0r" (for the path through which the CC78K0R is started). If the token string is replaced with "file name", the current folder is searched. If the specified file is not found, the CC78K0R searches in the folder specified with compiler option -i, folder specified by the INC78K0R environment variable, and then folder "..\inc78k0r" (for the path through which the CC78K0R is started).

9.3 Macro Replacement Directives

The macro replacement directives #define and #undef are used to replace the character string specified by "identifier" with "substitution list" and to end the scope of the identifier given by the #define, respectively. The #define directive has 2 forms: Object format and Function format:

- Object format

  #define directive

- Function format

  #define ( ) directive

(1) Actual argument replacement

Actual argument replacement is executed after the arguments in the function-form macro call are identified. If the # or ## preprocessing token is not prefixed to a parameter in the replacement list or if the ## preprocessing token does not follow any such parameter, all macros in the list will be expanded before replacement with the corresponding macro arguments.

(2) # operator

The # preprocessing token replaces the corresponding macro argument with a char string processing token. In other words, if this preprocessing token is prefixed to a parameter in the replacement list, the corresponding macro argument will be translated into a character or character string.

(3) ## operator

The ## preprocessing token concatenates the 2 tokens on either side of the ## symbol into 1 token. This concatenation will take place before the next macro expansion and the ## preprocessing token will be deleted after the concatenation. The token generated from this concatenation will undergo macro expansion if it has a macro name.

[Example of ## operation]

The above macro replacement directive will be expanded as follows:

```
printf ( "x" "1" "= %d , x" "2" "= %s" , x1 , x2 ) ;
```

The concatenated char string will look like this.

```
printf ( "x1 = %d , x2 = %s" , x1 , x2 ) ;
```

```
#include        <stdio.h>

#define debug ( s , t ) printf ( "x" #s "= %d , x" #t "= %s" , x##s , x##t
) ;

void    main ( void ) {
        int     x1 , x2 ;

        debug ( 1 , 2 ) ;
}
```

(4) Re-scanning and further replacement

The preprocessing token string resulting from replacement of macro parameters in the list will be scanned again, together with all remaining preprocessing tokens in the source file.

Macro names currently being (not including the remaining preprocessing tokens in the source file) replaced will not be replaced even if they are found during scanning of the replacement list.

(5) Scope of macro definition

A macro definition (#define directive) continues macro replacement until it encounters the corresponding #undef directive

#define directive

SYNTAX

```
#define identifier replacement-list new-line
```

FUNCTION

- The #define directive in its simplest form replaces the specified identifier (manifest) with a given replacement list (any character sequence that does not contain a new-line) whenever the same identifier appears in the source code after the definition by this directive.

EXAMPLE

```
#define PAI     3.1415
```

- In the above example, the identifier "PAI" will be replaced with "3.1415" whenever it appears in the source code after the definition by this directive.

#define ( ) directive

SYNTAX

```
#define identifier (dentifier-list) replacement-list new-line
```

FUNCTION

- The function-format #define directive that has the form "#define name (name, ..., name) replacement-list" replaces the identifier specified in the function format with a given replacement list (any character sequence that does not contain a new-line).

  The same identifier that appears after this directive is replaced with the replacement list. The function-format macro replacement can perform replacement including parameters.

EXAMPLE

```
#define F ( n ) ( n * n )

void    main ( void ) {
        int     i ;

        i = F ( 2 ) ;
}
```

- In the above example, #define directive will replace "F(2)" with "(2 * 2)" and thus the value of i will become 4.
- For the safety' sake, be sure to enclose the replacement list in parentheses, because unlike a function definition, this function-form macro is merely to replace a sequence of characters.

#undef directive

SYNTAX

```
#undef identifier new-line
```

FUNCTION

- The #undef directive undefines the given identifier. In other words, this directive ends the scope of the identifier that has been set by the corresponding #define directive.

EXAMPLE

```
#define F ( n ) ( n * n )
    :
#undef  F
```

- In the above example, #undef directive will invalidate the identifier "F" previously specified by "#define F(n) (n * n)".

9.4 Line Control Directive

The preprocessor directive for line control "#line" replaces the line number to be used by the C compiler in translation with the number specified in this directive.

If a string (character string) is given in addition to the number, the directive also replaces the source file name the C compiler has with the specified string.

[To change the line number]

To change the line number, the specification is made as follows.

0 and numbers larger than 32,767 cannot be specified.

```
#line   numeric-string  new-line
```

<Example>

```
#line   10
```

[To change the line number and the file name]

To change the line number and file name, the specification is made as follows.

```
#line   numeric-string  "character string"    new-line
```

<Example>

```
#line   10      "file1.c"
```

[To change using preprocessor token string]

In addition to the specifications above, the following specification can also be made. In this case, the specified preprocessor token string must be either one of the above 2 examples after all the replacement.

```
#line   preprocessing-token-string       new-line
```

<Example>

```
#define LINE_NUM        100

#line   LINE_NUM
```

9.5 #error Preprocess Directive

#error preprocess directive is a directive that outputs a message including the specified preprocessor tokens and incompletely terminates a compile.

This preprocessor is used to terminate a compile.

This preprocessor is specified as follows.

```
#error  "preprocessing-token-string"  new-line
```

[Example]

```
#if     __K0R__
    :
#else
#error  "not for 78K0R"
    :
#endif
```

- In this example, the macro name “__K0R__”, which indicates the device microcontroller that the CC78K0R has, is used. If the device is the 78K0R, the program between #if and #else is compiled. In the other cases, the program between #else and #endif is compiled, but the compile will be terminated with an error message "not for 78K0R" output by #error directive.

9.6 #pragma Directives

#pragma directive is a directive to instruct the compiler to operate in the compiler definition method.

In the CC78K0R, several #pragma directives to generate codes for the 78K0R.

For the details of #pragma directives, refer to "CHAPTER 11 EXTENDED FUNCTIONS".

[Example]

- In this example, #pragma NOP directive enables the description to directly output a NOP instruction in the C source.

```
#pragma NOP
```

9.7 Null Directives

Source lines that contain only the # character and white space are called null directives. Null directives are simply discarded during preprocessing. In other words, these directives have no effect on the compiler. The syntax of null directives is given below.

```
# new-line
```

9.8 Compiler-Defined Macro Names

In the CC78K0R, the following macro names have been defined.

Table 9-1 List of Macro Names

Macro Names	Explaination
__LINE__	Line number of the current source line (decimal constant)
__FILE__	Source file name (string literal)
__DATE__	Date the source file was compiled (string literal in the form of "M*mm dd yyyy*")
__TIME__	Time of day the source file was compiled (string literal in the form of "*hh* : *mm* : *ss*")
__STDC__	Decimal constant "1" that indicates the compliance with ANSI[Note] specification

Note ANSI is the acronym for American National Standards Institute

A #define or #undef preprocessor directive must not be applied to these macro name and defined identifiers. All the macro names of the compiler definition start with underscore followed by an uppercase character or the second underscore.

In addition to the above macro names, macro names indicating the microcontroller names of devices depending on the device subject to applied product development and macro names indicating device names are provided. To output the object code for the target device, these macro names must be specified by the option at compilation time or by the processor type in the C source.

- Macro name indicating the microcontroller names of devices

```
__K0R__
```

- Macro name indicating the device name

  "__" is added before the device type name and "_" is added after the device type name.

  <Example>

```
__F1166A0_ __F1166A0Y_
```

  Caution Describe English characters in uppercase.

  Remark The device type names are the same as the ones specified by the -c option. For the device type names, refer to the reference related to device files.

The CC78K0R has a macro name indicating the memory model.

Define the macro name as follows when specifying a memory model.

<When specifying small model>

```
#define __K0R_SMALL__   1
```

<When specifying medium model>

```
#define __K0R_MEDIUM__  1
```

<When specifying large model>

```
#define __K0R_LARGE__   1
```

The device type for compile is specified by adding the followings to the command line

"-c device type name"

<Example>

```
cc78k0r -cF1166A0Y prime.c
```

The device type does not need to be specified on compile by specifying it at the start of the C source program.

"#pragma PC (device type)"

<Example>

```
#pragma PC      ( F1166A0Y )
```

However, the followings can be described before "#pragma PC (device type)"

- Comment
- Preprocessor directives that do not generate definition/reference of variables nor functions.

CHAPTER 10 LIBRARY FUNCTIONS

C has no instructions to transfer (input or output) data to and from external sources (peripheral devices and equipment). This is because of the C language designer's intent to hold the functions of C to a minimum. However, for actually developing a system, I/O operations are requisite. Thus, the CC78K0R is provided with library functions to perform I/O operations.

The CC78K0R is provided with library functions such as I/O, character/memory manipulation, program control, and mathematical functions. This chapter describes the library functions provided to the CC78K0R.

10.1 Interface Between Functions

To use a library function, the function must be called. Calling a library function is carried out by a call instruction. The arguments and return value of a function are passed by a stack and a register, respectively.

However, the first argument is, if possible, also passed by the register

10.1.1 Arguments

Placing or removing arguments on or from the stack is performed by the caller (calling side). The callee (called side) only references the argument values.

However, when the argument is passed by the register, the callee directly refers to the register and copies the value of the argument to another register, if necessary.

Arguments are placed on the stack one by one in descending order from last to top if the argument is passed on the stack.

The minimum unit of data can be stacked is 16 bits. A data type larger than 16 bits is stacked in units of 16 bits one by one from its MSB. An 8-bit type data is extended to a 16-bit type data for stacking.

The following shows the list of the passing of the first argument.

The function interface (passing of argument and storing of return value) of the standard library is the same as that of normal function.

Table 10-1 List of Passing First Argument

Type of First Argument	Passing Method
1-byte, 2-byte integers	AX
near data pointer	AX
3-byte integer	AX, BC
4-byte integer	AX, BC
function pointer, far data pointer	AX, BC
Floating-point number (float type)	AX, BC
Floating-point number (double type)	Handled as float type
Others	Passed via a stack

Remark Of the types shown above, 1- to 4-byte integers include structures and unions.

10.1.2 Return values

The return value of a function is stored in units of 16 bits starting from its LSB in the direction from the register BC to the register DE. When returning a structure, the first address of the structure is stored in the register BC, DE.

The following shows the list of the storing of the return value. The method of storing return values is the same as that of normal function.

Table 10-2 List of Storing Return Value

Type of Return Value	Method of Storing
1 bit	CY
1-byte, 2-byte integers	BC
near pointer	BC
4-byte integer	BC (low-order), DE (high-order)
far pointer	BC (low-order), DE (high-order)
Floating-point number (float type)	BC (low-order), DE (high-order)
Floating-point number (double type)	Handled as float type
Structure	Copies the structure to return to the area specific to the function and stores the address to BC, DE

10.1.3 Saving registers to be used by individual libraries

Library that uses HL saves the registers it uses to a stack.

Each library that uses a saddr area saves the saddr area it uses to a stack. A stack area is used as a work area for each library.

An example of the procedure for passing arguments and return values (with small model or medium model specified) is shown below.

<Called function>

```
"long func ( int a , long b , char *c ) ;"
```

(a) Placing arguments on the stack (by the caller)

High-order 16 bits of arguments "c" and "b", low-order 16 bits of argument "b" are placed on the stack in the order named. a is passed by AX register.

(b) Calling func by call instruction (by the caller)

Return address is placed on the stack next to low-order 16 bits of argument "b" and control is transferred to the function func.

(c) Saving registers to be used within the function (by the callee)

If register HL is to be used, HL is placed on the stack.

(d) Placing the first argument passed by the register on the stack (by the callee)

(e) Processing func and storing the return value in registers (by the callee)

The low-order 16 bits of the return value "long" are stored in BC and the high-order 16 bits of the return value, in DE.

(f) Restoring the stored first argument (by the callee)

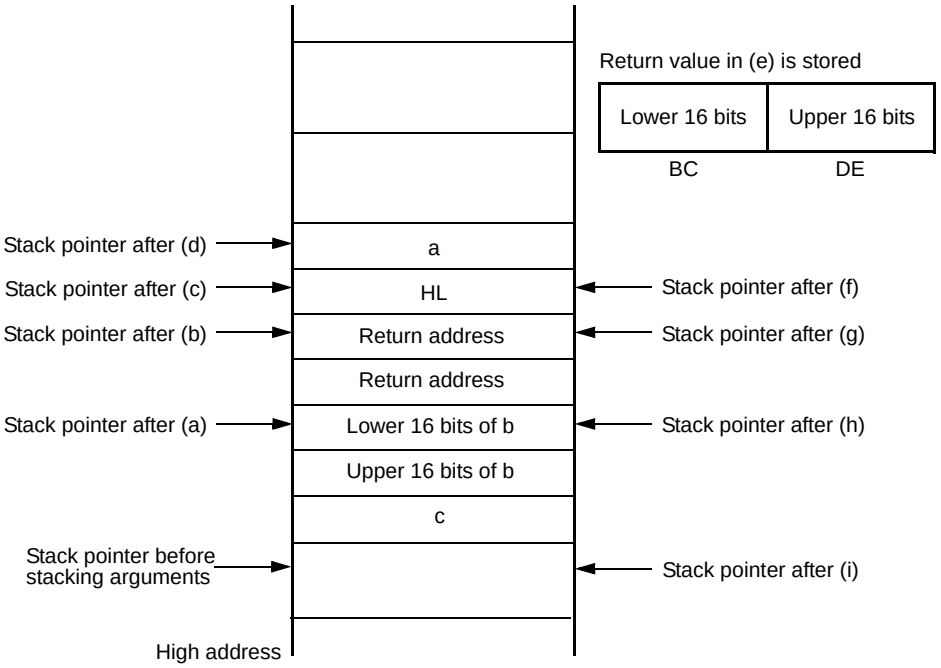
(g) Restoring the saved registers (by the callee)

(h) Returning control to the caller with ret instruction (by the callee)

(i) Removing arguments from the stack (by the caller)

The number of bytes (in units of 2 bytes) of the arguments is added to the stack pointer.

A value of 6 will be added to the stack pointer.



10.2 Headers

The CC78K0R has 13 headers (or header files). Each header defines or declares standard library functions, data type names, and macro names.

The headers of the CC78K0R are as shown below.

ctype.h	setjmp.h	stdarg.h	stdio.h	stdlib.h
string.h	error.h	errno.h	limits.h	stddef.h
math.h	float.h	assert.h		

(1) ctype.h

This header is used to define character and string functions.

In this standard header, the following library functions have been defined.

However, when compiler option -za (the option that disables the functions not complying ANSI specifications and enables a part of the functions of ANSI specifications) is specified, _toupper and _tolower are not defined. Instead, tolow and toup are defined. When the -za option is not specified, tolow and toup are not defined. The function to be declared differs depending on the options and the specification models.

isalnum	isalpha	iscntrl	isdigit	isgraph
islower	isprint	ispunct	isspace	isupper
isxdigit	tolower	toupper	isascii	toascii
_tolower	_toupper	tolow	toup	

(2) setjmp.h

This header is used to define program control functions.

In this header, the following functions are defined. The function to be declared differs depending on the option and the specification models.

setjmp	longjmp

In the header setjmp.h, the following object has been defined:

[Declaration of int array type jmp_buf]

```
typedef int     jmp_buf [ 12 ]
```

(3) stdarg.h

This header used to define special functions.

In this header, the following 3 functions have been defined:

```
va_arg          va_start        va_starttop     va_end
```

In the header stdarg.h the following object has been declared:

[Declaration of pointer type "va_list" to char]

```
typedef char    *va_list ;
```

(4) stdio.h

This header is used to define I/O functions. In this header, next functions have been defined.

The function to be declared differs depending on the options and the specification models.

```
sprintf         sscanf          printf          scanf           vprintf
vsprintf        getchar         gets            putchar         puts
__putc
```

The following macro names are declared.

```
#define EOF     ( -1 )
```

(5) stdlib.h

This header is used to define character and string functions, memory functions, program control functions, mathematical functions, and special functions. In this standard header, the following library functions have been defined:

However, when compiler option -za (the option that disables the functions not complying ANSI specifications and enables a part of the functions of ANSI specifications) is specified, brk, sbrk, itoa, ltoa, and ultoa are not defined. Instead, strbrk, strsbrk, stritoa, strltoa, and strultoa are defined. When the -za option is not specified, these functions are not defined.

```
atoi            atol            strtol          strtoul         calloc
free            malloc          realloc         abort           atexit
exit            abs             div             labs            ldiv
brk             sbrk            atof            strtod          itoa
ltoa            ultoa           rand            srand           bsearch
qsort           strbrk          strsbrk         stritoa         strltoa
strultoa
```

In the header stdlib.h the following objects have been defined:

[Declaration of structure type div_t which has int type members "quot" and "rem"]

```
typedef struct {
        int     quot ;
        int     rem ;
} div_t ;
```

[Definition of macro name "RAND_MAX"]

```
#define RAND_MAX        32767
```

[Declaration of macro name]

```
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    1
```

(6) string.h

This header is used to define character and string functions, memory functions, and special functions. In this header, the following functions have been defined. Function to be defined differs depending on the options and specification models.

```
memcpy          memmove         strcpy          strncpy         strcat
strncat         memcmp          strcmp          strncmp         memchr
strchr          strcspn         strpbrk         strrchr         strspn
strstr          strtok          memset          strerror        strlen
strcoll         strxfrm
```

(7) error.h

error.h includes errno.h.

(8) errno.h

In this header, the following objects have been defined:

[Definitions of macro names "EDOM", "ERANGE", and "ENOMEM"]

```
#define EDOM    1
#define ERANGE  2
#define ENOMEM  3
```

[Declaration of volatile int type external variable errno]

```
extern volatile int errno ;
```

(9) limits.h

In this header, the following macro names have been 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, which regards unqualified char as unsigned char, is specified, CHAR_MAX and CHAR_MIN are declared by the macro __CHAR_UNSIGNED__ declared by the compiler as follows.

```
#define CHAR_MAX        ( 255U )
#define CHAR_MIN        ( 0 )
```

(10) stddef.h

In this header, the following objects have been declared and defined:

[Declaration of int type "ptrdiff_t"]

```
typedef int     ptrdiff_t ;
```

[Declaration of unsigned int type "size_t"]

```
typedef unsigned int    size_t ;
```

[Definition of macro name "NULL"]

```
#define NULL    ( void * ) 0 ;
```

[Definition of macro name "offsetof"]

```
#define offsetof ( type , member )  ( (size_t) & (((type*)0) -> member) )
```

Remark offsetof (*type* , *member-specifier*)

offsetof is expanded to the general integer constant expression that has type size_t and the value is an offset value in byte units from the start of the structure (that is specified by the type) to the structure member (that is specified by the member specifier).

The member specifier must be the one that the result of evaluation of expression& (t. member specifier) becomes an address constant when static type t; is declared. When the specified member is a bit field, the operation will not be guaranteed.

(11) math.h

math.h defines the following functions.

```
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 macro name "HUGE_VAL"]

```
#define HUGE_VAL  DBL_MAX
```

(12) float.h

float.h defines the following objects.

When the size of a double type is 32 bits, the macro to be defined are sorted by the macro __DOUBLE_IS_32BITS__ declared by the compiler.

```
#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 */
```

(13) assert.h

assert.h defines the following function.

```
__assertfail
```

assert.h defines the following objects.

```
#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, if the assert.h header file references another macro, NDEBUG, which is not defined by the assert.h header file, and if NDEBUG is defined as a macro when the assert.h is captured to the source file, the assert.h header file simply declares the assert macro as the one given below and does not define __assertfail.

```
#define assert ( p )    ( ( void ) 0 )
```

10.3 Re-entrantability

Re-entrant is a state where a function called from a program can be consecutively called from another program.

The standard library of the CC78K0R does not use static area allowing re-entrantability. Therefore, data in the storage used by functions will not be destroyed by the call from another program.

However, the functions shown below are not re-entrant.

- Functions that cannot be re-entranced

```
setjmp, longjmp, atexit, exit
```

- Functions that uses the area secured in the startup routine

```
div, ldiv, brk, sbrk, rand, srand, strtok
```

- Functions that deals with floating point numbers

```
sprintf, sscanf, printf, scanf, vprintf, vsprintf[Note]
atof, strtod, all the mathematical functions
```

Note Among sprintf, sscanf, printf, scanf, vprintf, and vsprintf, ones that do not support floating-point numbers are re-entrant.

10.4 Standard Library Functions

This section explains the standard library functions of the CC78K0R by classifying them by function as follows.

All standard library functions are supported even when the -zf option is specified.

Table 10-3 List of Standard Library Functions

Type of Function	Function
Character/String Functions	is-
	toupper, tolower
	toascii
	_toupper/toup, _tolower/tolow
Program Control Functions	setjmp, longjmp
Special Functions	va_start, va_starttop, va_arg, va_end
I/O Functions	sprintf
	sscanf
	printf
	scanf
	vprintf
	vsprintf
	getchar
	gets
	putchar
	puts
	__putc

Table 10-3 List of Standard Library Functions

Type of Function	Function
Utility Functions	atoi, atol
	strtol, strtoul
	calloc
	free
	malloc
	realloc
	abort
	atexit, exit
	abs, labs
	div, ldiv
	brk, sbrk
	atof, strtod
Utility Functions	itoa, ltoa, ultoa
	rand, srand
	bsearch
	qsort
	strbrk
	strsbrk
	stritoa, strltoa, strultoa

Table 10-3 List of Standard Library Functions

Type of Function	Function
Character String/Memory Functions	memcpy, memmove
	strcpy, strncpy
	strcat, strncat
	memcmp
	strcmp, strncmp
	memchr
	strchr, strrchr
	strspn, strcspn
	strpbrk
	strstr
	strtok
	memset
	strerror
	strlen
	strcoll
	strxfrm
Mathematical Functions	acos
	asin
	atan
	atan2
	cos
	sin
	tan
	cosh
	sinh

Table 10-3 List of Standard Library Functions

Type of Function	Function
Mathematical Functions	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

Table 10-3 List of Standard Library Functions

Type of Function	Function
Mathematical Functions	ceilf
	fabsf
	floorf
	fmodf
Diagnostic Functions	__assertfail

10.4.1 Use of optimum library for arguments and return values

With the standard library that a pointer is specified to arguments and return values, an optimum library will be linked according to the memory model specified.
To handle a pointer that is not prepared as a default pointer of the memory model, call the function with the following standard function name; an optimum library for the pointer can then be linked.

<Function name>_n: The pointer is always handled as near pointer

<Function name>_f: The pointer is always handled as far pointer

For example, the far pointer can be specified as an argument of the strcmp function when the small model is used.

<Example>

```
#include <string.h>

__far char * sf1;
__far char * sf2;

void    main ( void ) {
          :
        r = strcmp_f ( sf1 , sf2 );
          :
}
```

[Cautions]

- When the small model or medium model is used, the pointer arguments of I/O functions sprintf, printf, vprintf, vsprintf, sscanf, and scanf, which treat variable arguments, are handled as near pointers. No function pointers can be used.
  When function pointers or far pointers are used, use printf_f, and all of the variable argument pointers must be converted to the far pointer.
  When the large model is used, the pointer arguments of I/O functions sprintf, printf, vprintf, vsprintf, sscanf, and scanf, which treat variable arguments, are handled as far pointers.
- When the small model or medium model is used, the pointer arguments of special functions va_start, va_starttop, va_arg, and va_end, which treat variable arguments, are handled as near pointers. No function pointers can be used.

10.5 Character/String Functions

The following character / string functions are available.

- is-
- toupper, tolower
- toascii
- _toupper/toup, _tolower/tolow

is-

FUNCTION

- is- judges the type of character.

HEADER

- ctype.h for all the character functions

FUNCTION PROTOTYPE

- int is- (int *c*) ;

Function	Arguments	Return Value
is-	*c*: Character to be judged	If character *c* is included in the character range: 1 If character *c* is not included in the character range: 0

EXPLANATION

Function	Character Range
isalpha	Alphabetic character A to Z or a to z
isupper	Uppercase letters A to Z
islower	Lowercase letters a to z
isdigit	Numeric characters 0 to 9
isalnum	Alphanumeric characters 0 to 9 and A to Z or a to z
isxdigit	Hexadecimal numbers 0 to 9 and A to F or a to f
isspace	White-space characters (space, tab, carriage return, new-line, vertical tab, and form-feed)
ispunct	Punctuation characters except white-space characters
isprint	Printable characters
isgraph	Printable nonblank characters
iscntrl	Control characters
isascii	ASCII code set

toupper, tolower

FUNCTION

- The character functions toupper and tolower both convert one type of character to another.
- The toupper function returns the uppercase equivalent of *c* if *c* is a lowercase letter.
- The tolower function returns the lowercase equivalent of *c* if *c* is a uppercase letter.

HEADER

- ctype.h

FUNCTION PROTOTYPE

- int toupper (int *c*) ;
- int tolower (int *c*) ;

Function	Arguments	Return Value
toupper, tolower	*c*: Character to be converted	If *c* is a convertible character: Uppercase equivalent If not convertible: Character "*c*" is returned unchanged

EXPLANATION

toupper

- The toupper function checks to see if the argument is a lowercase letter and if so converts the letter to its uppercase equivalent.

tolower

- The tolower function checks to see if the argument is a uppercase letter and if so converts the letter to its lowercase equivalent.

toascii

FUNCTION

- The character function toascii converts "*c*" to an ASCII code.

HEADER

- ctype.h

FUNCTION PROTOTYPE

- int toascii (int *c*) ;

Function	Arguments	Return Value
toascii	*c*: Character to be converted	Value obtained by converting the bits outside the ASCII code range of "*c*" to 0.

EXPLANATION

- 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/toup, _tolower/tolow

FUNCTION

- The character function _toupper/toup subtracts "a" from "*c*" and adds "A" to the result.
- The character function _tolower/tolow subtracts "A" from "*c*" and adds "a" to the result.
  (_toupper is exactly the same as toup, and _tolower is exactly the same as the tolow)

  Remark a: Lowercase; A: Uppercase

HEADER

- ctype.h

FUNCTION PROTOTYPE

- int _toupper/toup (int *c*) ;
- int _tolower/tolow (int *c*) ;

Function	Arguments	Return Value
_toupper/toup	*c*: Character to be converted	Value obtained by adding "A" to the result of subtraction "*c*" - "a"
_tolower/tolow		Value obtained by adding "a" to the result of subtraction "*c*" -"A"

Remark where a: Lowercase ;A: Uppercase

EXPLANATION

_toupper

- The _toupper function is similar to toupper except that it does not test to see if the argument is a lowercase letter.

_tolower

- The _tolower function is similar to tolower, except it does not test to see if the argument is an uppercase letter.

10.6 Program Control Functions

The following program control functions are available.

- setjmp, longjmp

setjmp, longjmp

FUNCTION

- The program control function setjmp saves the environment information (current state of the program) when a call to this function is made.
- The program control function longjmp restores the environment information saved by setjmp.

HEADER

- setjmp. h

FUNCTION PROTOTYPE

- int setjmp (jmp_buf *env*) ;
- void longjmp (jmp_buf *env* , int *val*) ;

Function	Arguments	Return Value
setjmp	*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.
longjmp	*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*".

EXPLANATION

setjmp

- 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

- 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).

10.7 Special Functions

The following special functions are available.

- va_start, va_starttop, va_arg, va_end

va_start, va_starttop, va_arg, va_end

FUNCTION

- The va_start function (macro) is used to start a variable argument list.
- The va_starttop function (macro) is used to set processing of the variable number of arguments.
- The va_arg function (macro) obtains the value of an argument from a variable argument list.
- The va_end function (macro) indicates that the end of a variable argument list is reached.

HEADER

- stdarg.h

FUNCTION PROTOTYPE

- void va_start (va_list *ap* , *parmN*) ;
- void va_starttop (va_list *ap* , *parmN*) ;
- type va_arg (va_list *ap* , *type*) ;
- void va_end (va_list *ap*) ;

Remark va_list is defined as typedef by stdarg.h.

Function	Arguments	Return Value
va_start, va_starttop	*ap*: Variable to be initialized so as to be used in va_arg and va_end *parmN* : The argument before variable argument	None
va_arg	*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
va_end	*ap*: Variable to process the variable number of arguments	None

EXPLANATION

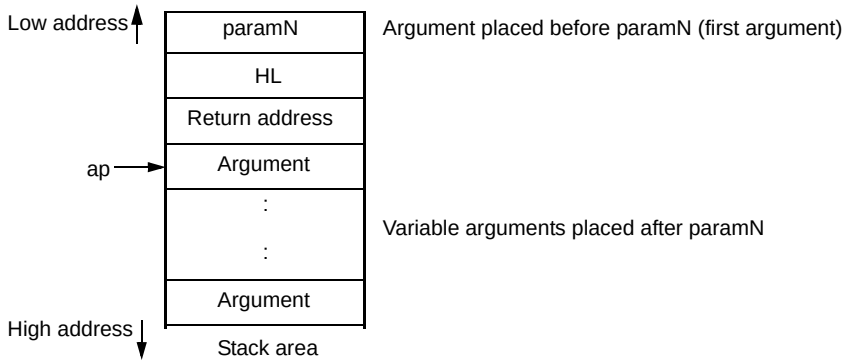
va_start

- 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

- *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

- 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 CC78K0R, 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

- 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.

10.8 I/O Functions

The following I/O functions are available.

- sprintf
- sscanf
- printf
- scanf
- vprintf
- vsprintf
- getchar
- gets
- putchar
- puts
- __putc

sprintf

FUNCTION

- The sprintf function writes data into a character string (array) according to the format.

HEADER

- stdio.h

FUNCTION PROTOTYPE

- int sprintf (char *s , const char *format , ...) ;

Function	Arguments	Return Value
sprintf	*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.)

EXPLANATION

- 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:

  (i) Zero or more flags (to be explained later) that modify the meaning of the format command

  (ii) 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 modifier 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.

The format codes for output conversion specifications are as follows:

Format Code	Contents
d	Converts int argument to signed decimal format.
i	Converts int argument to signed decimal format.
o	Converts int argument to unsigned octal format.
u	Converts int argument to unsigned decimal 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.

Precision Code	Contents
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 hexadecimal 4 digits (with 0s prefixed to less than a 4-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.

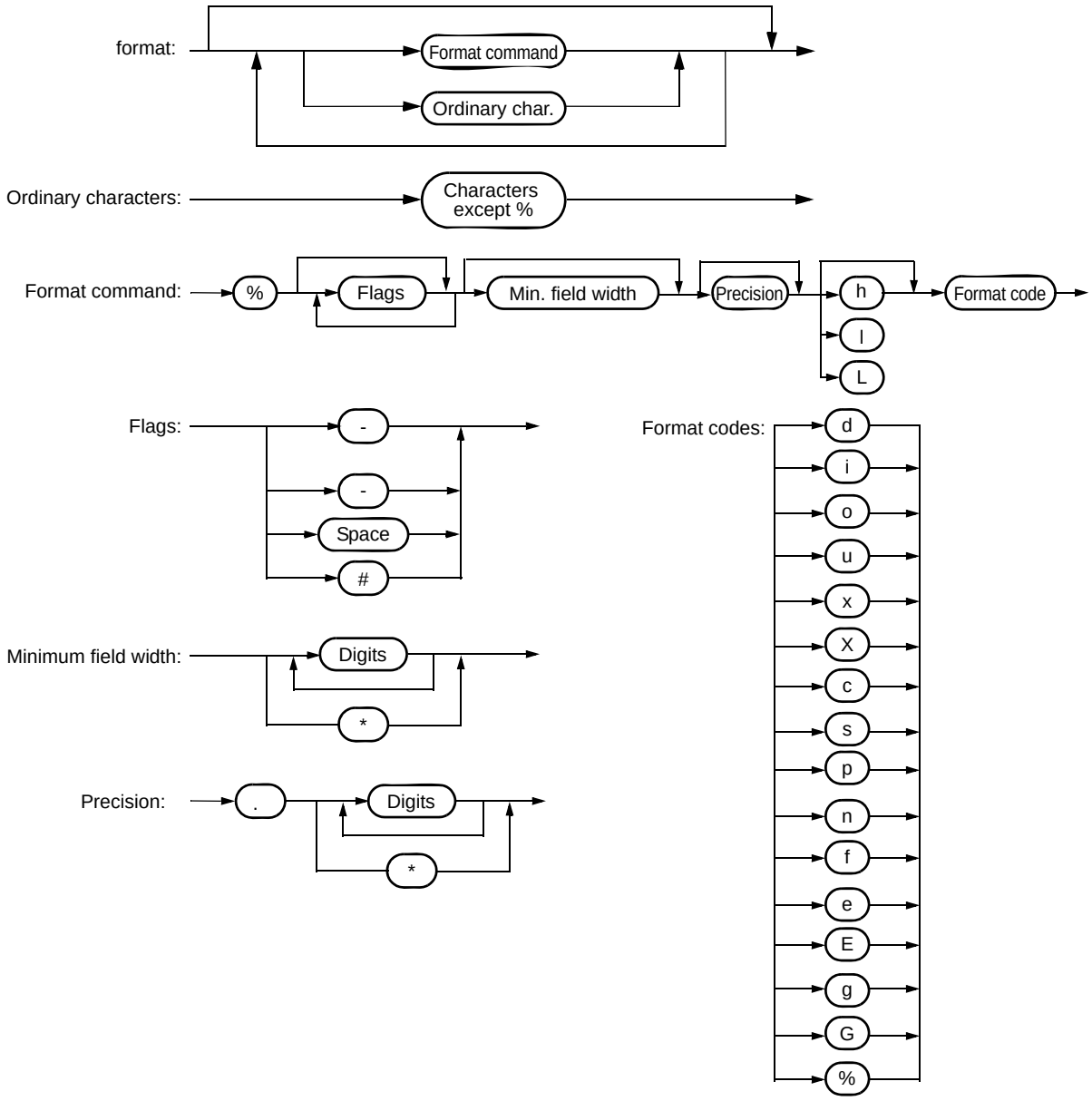
  non-numeric -> "(NaN)"

  +∞ -> "(+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.



- With the CC78K0R, 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

FUNCTION

- The sscanf function reads data from the input string (array) according to the format.

HEADER

- stdio.h

FUNCTION PROTOTYPE

- int sscanf (const char *s , const char *format , ...) ;

Function	Arguments	Return Value
sscanf	*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

EXPLANATION

- 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: White-space characters (one or more characters for which isspace becomes true)

  2: Non-white-space characters (other than %)

  3: Format specifiers

- Each format specifier begins with the % character and is followed by these:

  (i) Optional * character which suppresses assignment of data to the corresponding argument

  (ii) Optional decimal integer which specifies a maximum field width

  (iii) 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 white-space 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.

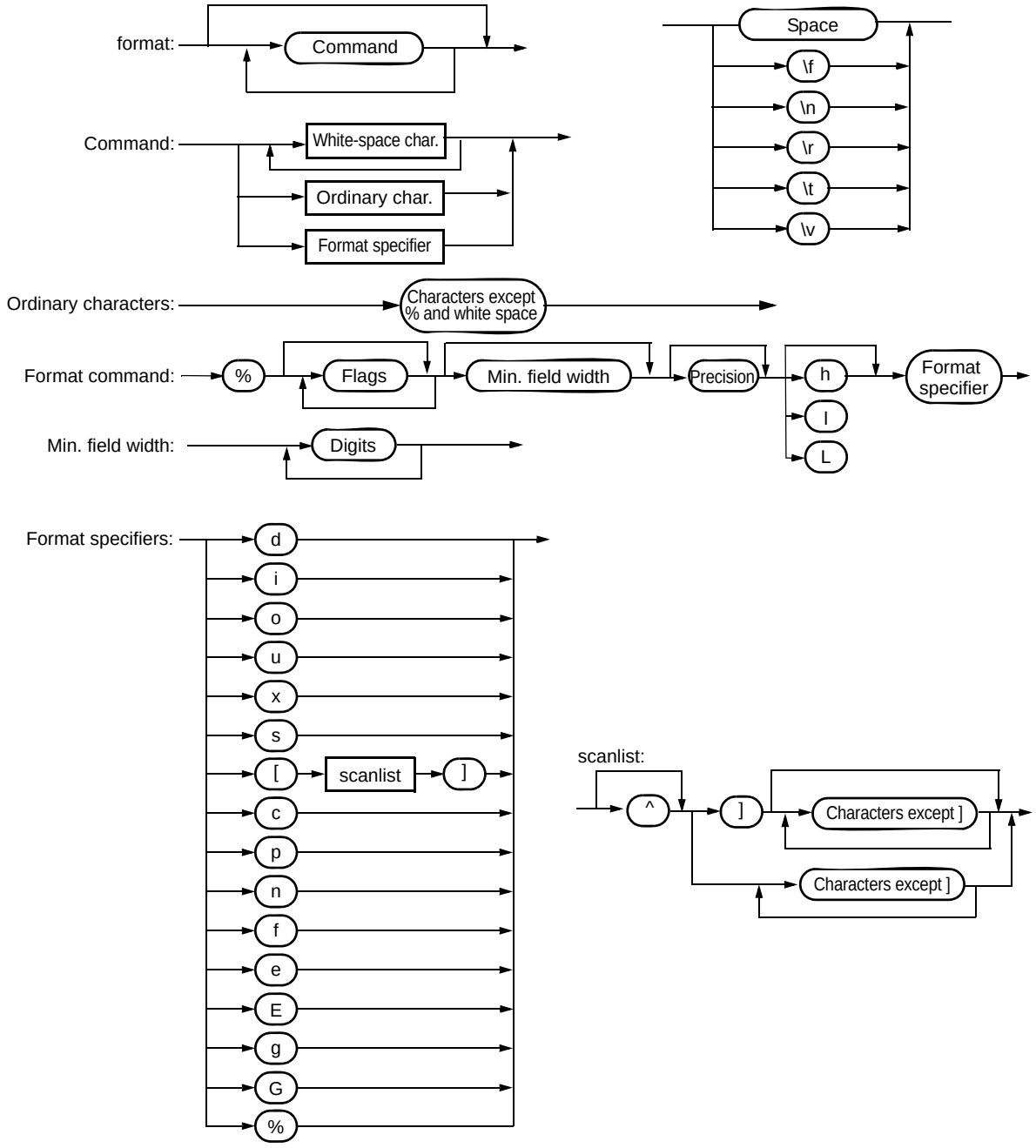
Conversion Specifier	Contents
s	Input a character string consisting of a non-blank character string. The corresponding argument is a pointer to an integer. 0x or 0X can be allocated at the first hexadecimal integer. The corresponding argument must be a pointer an array that has sufficient size to accommodate 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 (1). 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 input format commands is illustrated below.



- With the CC78K0R, 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

FUNCTION

- printf outputs data to SFR according to the format.

HEADER

- stdio.h

FUNCTION PROTOTYPE

- int printf (const char **format* , ...) ;

Function	Arguments	Return Value
printf	*format*: Pointer to the character string that indicates the output conversion specification ...: 0 or more arguments to be converted	Number of character output to *s* (the null character at the end is not counted)

EXPLANATION

- (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

FUNCTION

- scanf reads data from SFR according to the format.

HEADER

- stdio.h

FUNCTION PROTOTYPE

- int scanf (const char **format* , ...) ;

Function	Arguments	Return Value
scanf	*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

EXPLANATION

- 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

FUNCTION

- vprintf outputs data to SFR according to the format.

HEADER

- stdio.h

FUNCTION PROTOTYPE

- int vprintf (const char **format*, va_list *p*) ;

Function	Arguments	Return Value
vprintf	*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)

EXPLANATION

- 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

FUNCTION

- vsprintf writes data to character strings according to the format.

HEADER

- stdio.h

FUNCTION PROTOTYPE

- int vsprintf (char *s , const char *format, va_list p) ;

Function	Arguments	Return Value
vsprintf	*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)

EXPLANATION

- 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

FUNCTION

- getchar reads a character from SFR

HEADER

- stdio.h

FUNCTION PROTOTYPE

- int getchar (void) ;

Function	Arguments	Return Value
getchar	None	A character read from SFR

EXPLANATION

- 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

FUNCTION

- gets reads a character string.

HEADER

- stdio.h

FUNCTION PROTOTYPE

- char *gets (char *s) ;

Function	Arguments	Return Value
gets	*s*: Pointer to input character string	Normal: *s* If the end of the file is detected without reading a character: Null pointer

EXPLANATION

- 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

FUNCTION

- putchar outputs a character to SFR.

HEADER

- stdio.h

FUNCTION PROTOTYPE

- int putchar (int *c*) ;

Function	Arguments	Return Value
putchar	*c*: Character to be output	Character to have been output

EXPLANATION

- 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

FUNCTION

- puts outputs a character string.

HEADER

- stdio.h

FUNCTION PROTOTYPE

- int puts (const char *s) ;

Function	Arguments	Return Value
puts	*s*: Pointer to an output character string	Normal: 0 When putchar function returns -1: -1

EXPLANATION

- 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

FUNCTION

- __putc outputs a character to opaque.

HEADER

- stdio.h

FUNCTION PROTOTYPE

- int __putc (int *c* , void **opaque*) ;

Function	Arguments	Return Value
__putc	*c*: Character to be output *opaque*: Pointer to a character output destination	Character to have been output

EXPLANATION

- 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.

10.9 Utility Functions

The following utility functions are available.

- 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

atoi, atol

FUNCTION

- The string function atoi converts the contents of a decimal integer string to an int value.
- The string function atol converts the contents of a decimal integer string to a long int value.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- int atoi (const char *_nptr_) ;
- long int atol (const char *_nptr_) ;

Function	Arguments	Return Value
atoi	*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) If the string is invalid: 0
atol		If converted properly: long int value for positive overflow: LONG_MAX (2,147,483,647) for negative overflow: LONG_MIN (-2,147,483,648) If the string is invalid: 0

EXPLANATION

atoi

- 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

- 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, strtoul

FUNCTION

- The string function strtol converts a string to a long integer.
- The string function strtoul converts a string to an unsigned long integer.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- long int strtol (const char *_nptr_ , char **_endptr_, int _base_) ;
- unsigned long int strtoul (const char *_nptr_, char **_endptr_, int _base_) ;

Function	Arguments	Return Value
strtol	_nptr_: String to be converted _endptr_: Address of char pointer _base_: Base for number represented in the string	If converted properly: long int value for positive overflow: LONG_MAX (2,147,483,647) for negative overflow: LONG_MIN (-2,147,483,648) If not converted: 0
strtoul		If converted properly: unsigned long If overflow occurs: ULONG_MAX (4,294,967,295U) If not converted: 0

EXPLANATION

strtol

- The strtol function decomposes the string pointed by pointer *nptr* into the following 3 parts:

  (i) String of white-space characters that may be empty (to be specified by isspace)

  (ii) Integer representation by the *base* determined by the value of *base*

  (iii) String of one or more characters that cannot be recognized (including null terminators)

  Remark The strtol function converts the part (ii) 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 (iii) 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 (ii).
- If the string (ii) 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

- The strtoul function decomposes the string pointed by pointer *nptr* into the following 3 parts:

  (i) String of white-space characters that may be empty (to be specified by isspace)

  (ii) Integer representation by the *base* determined by the value of *base*

  (iii) String of one or more characters that cannot be recognized (including null terminators)

  The strtoul function converts the part (ii) 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 (iii) 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 (ii).
- If the string (ii) 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.

calloc

FUNCTION

- The memory function calloc allocates an array area and then initializes the area to 0.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- void *calloc (size_t *nmemb* , size_t *size*) ;

Function	Arguments	Return Value
calloc	*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

EXPLANATION

- 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, sbrk" for break value setting with the memory function brk.
- Since the areas to be allocated by the CC78K0R exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the calloc_n and calloc_f functions are not available.

free

FUNCTION

- The memory function free releases the allocated block of memory.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- void free (void *_ptr_) ;

Function	Arguments	Return Value
free	*ptr*: Pointer to the beginning of block to be released	None

EXPLANATION

- 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 CC78K0R exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the free_n and free_f functions are not available.

malloc

FUNCTION

- The memory function malloc allocates a block of memory.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- void *malloc (size_t *size*) ;

Function	Arguments	Return Value
malloc	*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

EXPLANATION

- 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 "10.9 Utility Functions brk, sbrk" for break value setting with the memory function brk.
- Since the areas to be allocated by the CC78K0R exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the malloc_n and malloc_f functions are not available.

realloc

FUNCTION

- The memory function realloc reallocates a block of memory (namely, changes the size of the allocated memory).

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- void *realloc (void **ptr* , size_t *size*) ;

Function	Arguments	Return Value
realloc	*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

EXPLANATION

- 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 CC78K0R exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the realloc_n and realloc_f functions are not available.

abort

FUNCTION

- The program control function abort causes immediate, abnormal termination of a program.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- void abort (void) ;

Function	Arguments	Return Value
abort	None	No return to its caller.

EXPLANATION

- The abort function loops and can never return to its caller.
- The user must create the abort processing routine.

atexit, exit

FUNCTION

- atexit registers the function called at the normal termination.
- exit terminates a program.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- int atexit (void (**func*) (void)) ;
- void exit (int *status*) ;

Function	Arguments	Return Value
atexit	*func*: Pointer to function to be registered	If function is registered as wrap-up function: 0 If function cannot be registered: 1
exit	*status*: Status value indicating termination	exit can never return.

EXPLANATION

atexit

- 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

- 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, labs

FUNCTION

- The mathematical function abs returns the absolute value of its int type argument.
- The mathematical function labs returns the absolute value of its long type argument.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- int abs (int *j*) ;
- long int labs (long int *j*) ;

Function	Arguments	Return Value
abs	*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)
labs		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: -2,147,483,648 (0x80000000)

EXPLANATION

abs

- The abs returns the absolute value of its int type argument.
- If *j* is -32,768, the function returns -32,768.

labs

- 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, ldiv

FUNCTION

- The mathematical function div performs the integer division of numerator divided by denominator.
- The mathematical function ldiv performs the long integer division of numerator divided by denominator.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- div_t div (int *numer* , int *denom*) ;
- ldiv_t ldiv (long int *numer* , long int *denom*) ;

Function	Arguments	Return Value
div	*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
ldiv		Quotient to the quot element and the remainder to the rem element of ldiv_t type member

EXPLANATION

div

- 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

- 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, sbrk

FUNCTION

- The memory function brk sets a break value.
- The memory function sbrk increments or decrements the set break value.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- int brk (char **endds*) ;
- char *sbrk (int *incr*) ;

Function	Arguments	Return Value
brk	*endds*: Break value to be set block to be released	If break value is set properly: 0 If break value cannot be changed: -1
sbrk	*incr*: Value (bytes) by which set break value is to be incremented/decremented.	If incremented or decremented properly: Old break value If old break value cannot be incremented or decremented: -1

EXPLANATION

brk

- 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 CC78K0R exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the brk_n and brk_f functions are not available.

sbrk

- 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 CC78K0R exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the sbrk_n and sbrk_f functions are not available.

atof, strtod

FUNCTION

- The string function atof converts the contents of a decimal integer string to a double value.
- The string function strtod converts the contents of a string to a double value.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- double atof (const char **nptr*) ;
- double strtod (const char **nptr* , char ***endptr*) ;

Function	Arguments	Return Value
atof	*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 If the string is invalid: 0
strtod	*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 If the string is invalid: 0

EXPLANATION

atof

- The atof function converts the string pointed to by pointer *nptr* to a double value.
- The atof 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 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

- 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, ltoa, ultoa

FUNCTION

- The string function itoa converts an int integer to its string equivalent.
- The string function ltoa converts a long int integer to its string equivalent.
- The string function ultoa converts an unsigned long integer to its string equivalent.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- char *itoa (int *value* , char **string* , int *radix*) ;
- char *ltoa (long *value* , char **string* , int *radix*) ;
- char *ultoa (unsigned long *value* , char **string* , int *radix*) ;

Function	Arguments	Return Value
itoa, ltoa, ultoa	*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

EXPLANATION

itoa, ltoa, ultoa

- 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, srand

FUNCTION

- The mathematical function rand generates a sequence of pseudorandom numbers.
- The mathematical function srand sets a starting value (seed) for the sequence generated by rand.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- int rand (void) ;
- void srand (unsigned int *seed*) ;

Function	Arguments	Return Value
rand	None	Pseudorandom integer in the range of 0 to RAND_MAX
srand	*seed*: Starting value for pseudorandom number generator	None

EXPLANATION

rand

- Each time the rand function is called, it returns a pseudorandom integer in the range of 0 to RAND_MAX.

srand

- 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

FUNCTION

- The bsearch function performs a binary search.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- void *bsearch (const void **key* , const void **base* , size_t *nmemb* , size_t *size* ,
  int (**compare*) (const void *, const void *)) ;

Function	Arguments	Return Value
bsearch	*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

EXPLANATION

- 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

FUNCTION

- The qsort function sorts the members of a specified array using a quicksort algorithm.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- void qsort (void *_base_ , size_t _nmemb_ , size_t _size_ , int (*_compare_) (const void * , const void *)) ;

Function	Arguments	Return Value
qsort	*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

EXPLANATION

- 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

FUNCTION

- strbrk sets a break value.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- int strbrk (char *_endds_) ;

Function	Arguments	Return Value
strbrk	*endds*: Break value to set	Normal: 0 When a break value cannot be changed: -1

EXPLANATION

- 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 CC78K0R exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the strbrk_n and strbrk_f functions are not available.

strsbrk

FUNCTION

- strsbrk increases/decreases a break value.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- char *strsbrk(int *incr*);

Function	Arguments	Return Value
strsbrk	*incr*: Amount to increase/decrease a break value	Normal: Old break value When a break value cannot be increased/ decreased: -1

EXPLANATION

- *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 CC78K0R exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the strsbrk_n and strsbrk_f functions are not available.

stritoa, strltoa, strultoa

FUNCTION

- stritoa converts int to a character string.
- strltoa converts long to a character string.
- strultoa converts unsigned long to a character string.

HEADER

- stdlib.h

FUNCTION PROTOTYPE

- char *stritoa (int *value* , char **string* , int *radix*) ;
- char *strltoa (long *value* , char **string* , int *radix*) ;
- char *strultoa (unsigned long *value* , char **string* , int *radix*) ;

Function	Arguments	Return Value
stritoa, strltoa, strultoa	*value*: Character string to convert *string*: Pointer to conversion result *radix*: Radix to specify	Normal: Pointer to the converted character string Others: Null pointer

EXPLANATION

- 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.

10.10 Character String/Memory Functions

The following character string/memory functions are available.

- memcpy, memmove
- strcpy, strncpy
- strcat, strncat
- memcmp
- strcmp, strncmp
- memchr
- strchr, strrchr
- strspn, strcspn
- strpbrk
- strstr
- strtok
- memset
- strerror
- strlen
- strcoll
- strxfrm

memcpy, memmove

FUNCTION

- The memory function memcpy copies a specified number of characters from a source area of memory to a destination area of memory.
- The memory function memmove is identical to memcpy, except that it allows overlap between the source and destination areas.

HEADER

- string.h

FUNCTION PROTOTYPE

- void *memcpy (void **s1*, const void **s2* , size_t *n*) ;
- void *memmove (void **s1*, const void **s2*, size_t *n*) ;

Function	Arguments	Return Value
memcpy, memmove	*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*

EXPLANATION

memcpy

- 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

- 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, strncpy

FUNCTION

- The string function strcpy is used to copy the contents of one character string to another.
- The string function strncpy is used to copy up to a specified number of characters from one character string to another.

HEADER

- string.h

FUNCTION PROTOTYPE

- char *strcpy (char **s1* , const char **s2*) ;
- char *strncpy (char **s1* , const char **s2* , size_t *n*) ;

Function	Arguments	Return Value
strcpy	*s1*: Pointer to copy destination array *s2*: Pointer to copy source array	Value of *s1*
strncpy	*s1*: Pointer to copy destination array *s2*: Pointer to copy source array *n*: Number of characters to be copied	Value of *s1*

EXPLANATION

strcpy

- 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

- 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, strncat

FUNCTION

- The string function strcat concatenates one character string to another.
- The string function strncat concatenates up to a specified number of characters from one character string to another.

HEADER

- string.h

FUNCTION PROTOTYPE

- char *strcat (char **s1* , const char **s2*) ;
- char *strncat (char **s1* , const char **s2* , size_t *n*) ;

Function	Arguments	Return Value
strcat	*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*
strncat	*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*

EXPLANATION

strcat

- 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

- 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

FUNCTION

- The memory function memcmp compares 2 data objects, with respect to a given number of characters.

HEADER

- string.h

FUNCTION PROTOTYPE

- int memcmp (const void *_s1_ , const void *_s2_ , size_t _n_) ;

Function	Arguments	Return Value
memcmp	_s1_, _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)

EXPLANATION

- 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, strncmp

FUNCTION

- The string function strcmp compares 2 character strings.
- The string function strncmp compares not more than a specified number of characters from 2 character strings.

HEADER

- string.h

FUNCTION PROTOTYPE

- char *strcmp (char **s1* , const char **s2*) ;
- char *strncmp (char **s1* , const char **s2* , size_t *n*) ;

Function	Arguments	Return Value
strcmp	*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)
strncmp	*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)

EXPLANATION

strcmp

- 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

- 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

FUNCTION

- The memory function memchr converts a specified character to unsigned char, searches for it, and returns a pointer to the first occurrence of this character in an object of a given size.

HEADER

- string.h

FUNCTION PROTOTYPE

- void *memchr (const void *s , int *c* , size_t *n*) ;

Function	Arguments	Return Value
memchr	*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

EXPLANATION

- 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, strrchr

FUNCTION

- The string function strchr returns a pointer to the first occurrence of a specified character in a string.
- The string function strrchr returns a pointer to the last occurrence of a specified character in a string.

HEADER

- string.h

FUNCTION PROTOTYPE

- char *strchr (const char *s , int c) ;
- char *strrchr (const char *s , int c) ;

Function	Arguments	Return Value
strchr, strrchr	*s*: Pointer to string to be searched *c*: Character specified for search	If *c* is found in *s*: Pointer indicating the first or last occurrence of *c* in string *s* If *c* is not found in *s*: Null pointer

EXPLANATION

strchr

- 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

- 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, strcspn

FUNCTION

- The string function strspn returns the length of the initial substring of a string that is made up of only those characters contained in another string.
- The string function strcspn returns the length of the initial substring of a string that is made up of only those characters not contained in another string.

HEADER

- string.h

FUNCTION PROTOTYPE

- size_t strspn (const char **s1* , const char **s2*) ;
- size_t strcspn (const char **s1* , const char **s2*) ;

Function	Arguments	Return Value
strspn	*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*
strcspn		Length of substring of the string *s1* that is made up of only those characters not contained in the *s2*

EXPLANATION

strspn

- 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

- 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

FUNCTION

- The string function strpbrk returns a pointer to the first character in a string to be searched that matches any character in a specified string.

HEADER

- string.h

FUNCTION PROTOTYPE

- char *strpbrk (const char **s1* , const char **s2*) ;

Function	Arguments	Return Value
strpbrk	*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

EXPLANATION

- 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

FUNCTION

- The string function strstr returns a pointer to the first occurrence in the string to be searched of a specified string.

HEADER

- string.h

FUNCTION PROTOTYPE

- char *strstr (const char **s1* , const char **s2*) ;

Function	Arguments	Return Value
strstr	*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*

EXPLANATION

- 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

FUNCTION

- The string function strtok returns a pointer to a token taken from a string (by decomposing it into a string consisting of characters other than delimiters).

HEADER

- string.h

FUNCTION PROTOTYPE

- char *strtok (char **s1* , const char **s2*) ;

Function	Arguments	Return Value
strtok	*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

EXPLANATION

- 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

FUNCTION

- The memory function memset initializes a specified number of bytes in an object in memory with a specified character.

HEADER

- string.h

FUNCTION PROTOTYPE

- void *memset (void *s , int c , size_t n) ;

Function	Arguments	Return Value
memset	*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*

EXPLANATION

- 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

FUNCTION

- The strerror function returns a pointer to the location which stores a string describing the error message associated with a given error number.

HEADER

- string.h

FUNCTION PROTOTYPE

- char *strerror (int *errnum*) ;

Function	Arguments	Return Value
strerror	*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

EXPLANATION

- The strerror function returns the following values associated with the value of *errnum*.

Value of *errnum*	Return Values
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

strlen

FUNCTION

- The string function strlen returns the length of a character string.

HEADER

- string.h

FUNCTION PROTOTYPE

- size_t strlen (const char *s) ;

Function	Arguments	Return Value
strlen	*s*: Pointer to character string	Length of string *s*

EXPLANATION

- The strlen function returns the length of the null terminated string pointed to by *s*.

strcoll

FUNCTION

- strcoll compares 2 character strings based on the information specific to the area.

HEADER

- string.h

FUNCTION PROTOTYPE

- int strcoll (const char **s1* , const char **s2*) ;

Function	Arguments	Return Value
strcoll	*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*)

EXPLANATION

- The CC78K0R does not support operations specific to cultural sphere.

  The operations are the same as that of strcmp.

strxfrm

FUNCTION

- strxfrm converts a character string based on the information specific to the area.

HEADER

- string.h

FUNCTION

- size_t strxfrm (char *s1 , const char *s2 , size_t *n*) ;

Function	Arguments	Return Value
strxfrm	*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.

EXPLANATION

- The CC78K0R 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 ) ) ;
```

10.11 Mathematical Functions

The following mathematical functions are available.

- 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

acos

FUNCTION

- acos finds acos.

HEADER

- math.h

FUNCTION PROTOTYPE

- double acos (double *x*) ;

Function	Arguments	Return Value
acos	*x*: Numeric value to perform operation	When $-1 \leq x \leq 1$: acos of *x* When $x < -1$, $1 < x$, x = NaN: NaN

EXPLANATION

- 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

FUNCTION

- asin finds asin.

HEADER

- math.h

FUNCTION PROTOTYPE

- double asin (double *x*) ;

Function	Arguments	Return Value
asin	*x*: Numeric value to perform operation	When $-1 \leq x \leq 1$: asin of *x* When $x < -1$, $1 < x$, x = NaN: NaN When x = -0: -0 When underflow occurs: Non-normalized number

EXPLANATION

- 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

FUNCTION

- atan finds atan.

HEADER

- math.h

FUNCTION PROTOTYPE

- double atan (double *x*) ;

Function	Arguments	Return Value
atan	*x*: Numeric value to perform operation	Normal: atan of *x* When *x* = NaN: NaN When *x* = -0: -0 When underflow occurs: Non-normalized number

EXPLANATION

- 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

FUNCTION

- atan2 finds atan of y/x.

HEADER

- math.h

FUNCTION PROTOTYPE

- double atan2 (double *y* , double *x*) ;

Function	Arguments	Return Value
atan2	*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

EXPLANATION

- 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

FUNCTION

- cos finds cos.

HEADER

- math.h

FUNCTION PROTOTYPE

- double cos (double *x*) ;

Function	Arguments	Return Value
cos	*x*: Numeric value to perform operation	Normal: cos of *x* When *x* = NaN, when *x* is infinite: NaN

EXPLANATION

- 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

FUNCTION

- sin finds sin.

HEADER

- math.h

FUNCTION PROTOTYPE

- double sin (double *x*) ;

Function	Arguments	Return Value
sin	*x*: Numeric value to perform operation	Normal: sin of *x* When *x* = NaN, when *x* is infinite: NaN When underflow occurs: Non-normalized number

EXPLANATION

- 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

FUNCTION

- tan finds tan.

HEADER

- math.h

FUNCTION PROTOTYPE

- double tan (double *x*) ;

Function	Arguments	Return Value
tan	*x*: Numeric value to perform operation	Normal: tan of *x* When *x* = NaN, *x* = ±∞: NaN When underflow occurs: Non-normalized number

EXPLANATION

- 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

FUNCTION

- cosh finds cosh.

HEADER

- math.h

FUNCTION PROTOTYPE

- double cosh (double *x*) ;

Function	Arguments	Return Value
cosh	*x*: Numeric value to perform operation	Normal: cosh of *x* *x* = NaN: NaN *x* = ±∞: +∞ When overflow occurs HUGE_VAL (with the sign of the overflown value)

EXPLANATION

- 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

FUNCTION

- sinh finds sinh.

HEADER

- math.h

FUNCTION PROTOTYPE

- double sinh (double *x*) ;

Function	Arguments	Return Value
sinh	*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

EXPLANATION

- 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

FUNCTION

- tanh finds tanh.

HEADER

- math.h

FUNCTION PROTOTYPE

- double tanh (double *x*) ;

Function	Arguments	Return Value
tanh	*x*: Numeric value to perform operation	Normal: tanh of *x* When *x* = NaN: NaN When *x* = ±∞: ±1 When underflow occurs: ±0

EXPLANATION

- 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

FUNCTION

- exp finds exponent function.

HEADER

- math.h

FUNCTION PROTOTYPE

- double exp (double *x*) ;

Function	Arguments	Return Value
exp	*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)

EXPLANATION

- 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

FUNCTION

- frexp finds mantissa and exponent part.

HEADER

- math.h

FUNCTION PROTOTYPE

- double frexp (double *x* , int **exp*) ;

Function	Arguments	Return Value
frexp	*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

EXPLANATION

- 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

FUNCTION

- ldexp finds x * 2 ^ exp.

HEADER

- math.h

FUNCTION PROTOTYPE

- double ldexp (double *x* , int *exp*) ;

Function	Arguments	Return Value
ldexp	*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

EXPLANATION

- 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

FUNCTION

- log finds natural logarithm.

HEADER

- math.h

FUNCTION PROTOTYPE

- double log (double *x*) ;

Function	Arguments	Return Value
log	*x*: Numeric value to perform operation	Normal: Natural logarithm of *x* When *x* < 0: NaN When *x* = 0: -∞ When *x* is non-numeric: NaN When *x* is infinite: +∞

EXPLANATION

- 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

FUNCTION

- log10 finds logarithm with 10 as the base.

HEADER

- math.h

FUNCTION PROTOTYPE

- double log10 (double *x*) ;

Function	Arguments	Return Value
log10	*x*: Numeric value to perform operation	Normal: Logarithm with 10 of *x* as the base When *x* < 0: NaN When *x* = 0: -∞ When *x* is non-numeric: NaN When *x* is infinite: +∞

EXPLANATION

- 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

FUNCTION

- modf finds fraction part and integer part.

HEADER

- math.h

FUNCTION PROTOTYPE

- double modf (double *x* , double **iptr*) ;

Function	Arguments	Return Value
modf	*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

EXPLANATION

- 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

FUNCTION

- pow finds yth power of *x*.

HEADER

- math.h

FUNCTION PROTOTYPE

- double pow (double *x* , double *y*) ;

Function	Arguments	Return Value
pow	*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

EXPLANATION

- 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

FUNCTION

- sqrt finds square root.

HEADER

- math.h

FUNCTION PROTOTYPE

- double sqrt (double *x*) ;

Function	Arguments	Return Value
sqrt	*x*: Numeric value to perform operation	When $x \geq 0$: Square root of *x* When $x < 0$: 0 When *x* = NaN: NaN When $x = \pm 0$: ± 0

EXPLANATION

- 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

FUNCTION

- ceil finds the minimum integer no less than x.

HEADER

- math.h

FUNCTION PROTOTYPE

- double ceil (double *x*) ;

Function	Arguments	Return Value
ceil	*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*

EXPLANATION

- 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

FUNCTION

- fabs returns the absolute value of the floating point number x .

HEADER

- math.h

FUNCTION PROTOTYPE

- double fabs (double *x*) ;

Function	Arguments	Return Value
fabs	*x*: Numeric value to find the absolute value	Normal: Absolute value of *x* When *x* is non-numeric: NaN When *x* = -0: +0

EXPLANATION

- Finds the absolute value of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is -0, +0 is returned.

floor

FUNCTION

- floor finds the maximum integer no more than x.

HEADER

- math.h

FUNCTION PROTOTYPE

- double floor (double *x*) ;

Function	Arguments	Return Value
floor	*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*

EXPLANATION

- 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

FUNCTION

- fmod finds the remainder of x/y.

HEADER

- math.h

FUNCTION PROTOTYPE

- double fmod (double *x* , double *y*) ;

Function	Arguments	Return Value
fmod	*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*

EXPLANATION

- 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

FUNCTION

- matherr performs exception processing of the library that deals with floating point numbers.

HEADER

- math.h

FUNCTION PROTOTYPE

- void matherr (struct *exception *x*) ;

Function	Arguments	Return Value

matherr	`struct exception {`
	`  int  type;`
	`  char  *name;`
	`}`
	
	type:
	Numeric value to indicate arithmetic exception
	name:
	Function name
		None

EXPLANATION

- 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.

acosf

FUNCTION

- acosf finds acos.

HEADER

- math.h

FUNCTION PROTOTYPE

- float acosf (float *x*) ;

Function	Arguments	Return Value
acosf	*x*: Numeric value to perform operation	When $-1 \leq x \leq 1$: acos of *x* When $x < -1$, $1 < x$, x = NaN: NaN

EXPLANATION

- 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

FUNCTION

- asinf finds asin.

HEADER

- math.h

FUNCTION PROTOTYPE

- float asinf (float *x*) ;

Function	Arguments	Return Value
asinf	*x*: Numeric value to perform operation	When $-1 \leq x \leq 1$: asin of *x* When $x < -1$, $1 < x$, *x* = NaN: NaN *x* = -0: -0 When underflow occurs: Non-normalized number

EXPLANATION

- 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

FUNCTION

- atanf finds atan.

HEADER

- math.h

FUNCTION PROTOTYPE

- float atanf (float *x*) ;

Function	Arguments	Return Value
atanf	*x*: Numeric value to perform operation	Normal: atan of *x* When *x* = NaN: NaN When *x* = -0: -0 When underflow occurs: Non-normalized number

EXPLANATION

- 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

FUNCTION

- atan2f finds atan of y/x.

HEADER

- math.h

FUNCTION PROTOTYPE

- float atan21 (float *y* , float *x*) ;

Function	Arguments	Return Value
atan2f	*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

EXPLANATION

- Calculates atan (range between -π and +π) 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

FUNCTION

- cosf finds cos.

HEADER

- math.h

FUNCTION PROTOTYPE

- float cosf (float *x*) ;

Function	Arguments	Return Value
cosf	*x*: Numeric value to perform operation	Normal: cos of *x* When *x* = NaN, *x* is infinite: NaN

EXPLANATION

- 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

FUNCTION

- sinf finds sin.

HEADER

- math.h

FUNCTION PROTOTYPE

- float sinf (float *x*) ;

Function	Arguments	Return Value
sinf	*x*: Numeric value to perform operation	Normal: sin of *x* When *x* = NaN, when *x* is infinite: NaN When underflow occurs: Non-normalized number

EXPLANATION

- 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

FUNCTION

- tanf finds tan.

HEADER

- math.h

FUNCTION PROTOTYPE

- float tanf (float *x*) ;

Function	Arguments	Return Value
tanf	*x*: Numeric value to perform operation	Normal: tan of *x* When *x* = NaN, when *x* is infinite: NaN When underflow occurs: Non-normalized number

EXPLANATION

- 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

FUNCTION

- coshf finds cosh.

HEADER

- math.h

FUNCTION PROTOTYPE

- float coshf (float *x*) ;

Function	Arguments	Return Value
coshf	*x*: Numeric value to perform operation	Normal: cosh of *x* *x* = NaN: NaN When *x* is infinite: +∞ When overflow occurs: HUGE_VAL (with a positive sign)

EXPLANATION

- 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

FUNCTION

- sinhf finds sinh.

HEADER

- math.h

FUNCTION PROTOTYPE

- float sinhf (float *x*) ;

Function	Arguments	Return Value
sinhf	*x*: Numeric value to perform operation	Normal: sinh of *x* *x* = NaN: NaN When *x* = ±∞: ±∞ When overflow occurs: HUGE_VAL (with a sign of the overflown value) When underflow occurs: ±0

EXPLANATION

- 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

FUNCTION

- tanhf finds tanh.

HEADER

- math.h

FUNCTION PROTOTYPE

- float tanhf (float *x*) ;

Function	Arguments	Return Value
tanhf	*x*: Numeric value to perform operation	Normal: tanh of *x* *x* = NaN: NaN When *x* = ±∞: ±1 When underflow occurs: ±0

EXPLANATION

- 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

FUNCTION

- expf finds exponent function.

HEADER

- math.h

FUNCTION PROTOTYPE

- float expf (float *x*) ;

Function	Arguments	Return Value
expf	*x*: Numeric value to perform operation	Normal: Exponent function of *x* *x* = NaN: NaN When *x* = +∞: +∞ When *x* = -∞: +0 When overflow occurs: HUGE_VAL (with positive sign) When underflow occurs: Non-normalized number When annihilation of effective digits occurs due to underflow: +0

EXPLANATION

- 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

FUNCTION

- frexpf finds mantissa and exponent part.

HEADER

- math.h

FUNCTION PROTOTYPE

- float frexpf (float *x* , int **exp*) ;

Function	Arguments	Return Value
frexpf	*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

EXPLANATION

- 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

FUNCTION

- ldexpf finds x * 2 ^ exp.

HEADER

- math.h

FUNCTION PROTOTYPE

- float ldexpf (float *x* , int *exp*) ;

Function	Arguments	Return Value
ldexpf	*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

EXPLANATION

- 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

FUNCTION

- logf finds natural logarithm.

HEADER

- math.h

FUNCTION PROTOTYPE

- float logf (float *x*) ;

Function	Arguments	Return Value
logf	*x*: Numeric value to perform operation	Normal: Natural logarithm of *x* When *x* < 0: NaN When *x* = 0: -∞ When *x* is non-numeric: NaN When *x* is infinite: +∞

EXPLANATION

- 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

FUNCTION

- log10f finds logarithm with 10 as the base.

HEADER

- math.h

FUNCTION PROTOTYPE

- float log10f (float *x*) ;

Function	Arguments	Return Value
log10f	*x*: Numeric value to perform operation	Normal: Logarithm with 10 of *x* as the base When *x* < 0: NaN When *x* = 0: -∞ When *x* is non-numeric: NaN When *x* = +∞: +∞

EXPLANATION

- 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

FUNCTION

- modff finds fraction part and integer part.

HEADER

- math.h

FUNCTION PROTOTYPE

- float modff (float *x* , float **iptr*) ;

Function	Arguments	Return Value
modff	*x*: Numeric value to perform operation *iptr*: Pointer for integer part	Normal: Fraction part of *x* When *x* is non-numeric or infinite: NaN When *x* = ±0: ±0

EXPLANATION

- 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

FUNCTION

- powf finds yth power of *x*.

HEADER

- math.h

FUNCTION PROTOTYPE

- float powf (float *x* , float *y*) ;

Function	Arguments	Return Value
powf	*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

EXPLANATION

- 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

FUNCTION

- sqrtf finds square root.

HEADER

- math.h

FUNCTION PROTOTYPE

- float sqrtf (float *x*) ;

Function	Arguments	Return Value
sqrtf	*x*: Numeric value to perform operation	When $x \geq 0$: Square root of *x* When $x < 0$: 0 When *x* = NaN: NaN When $x = \pm 0$: ± 0

EXPLANATION

- 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

FUNCTION

- ceilf finds the minimum integer no less than x.

HEADER

- math.h

FUNCTION PROTOTYPE

- float ceilf (float *x*) ;

Function	Arguments	Return Value
ceilf	*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*

EXPLANATION

- 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

FUNCTION

- fabsf returns the absolute value of the floating point number x.

HEADER

- math.h

FUNCTION PROTOTYPE

- float fabsf (float *x*) ;

Function	Arguments	Return Value
fabsf	*x*: Numeric value to find the absolute value	Normal: Absolute value of *x* When *x* is non-numeric: NaN When *x* = -0: +0

EXPLANATION

- Finds the absolute value of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is -0, +0 is returned.

floorf

FUNCTION

- floorf finds the maximum integer no more than x.

HEADER

- math.h

FUNCTION PROTOTYPE

- float floorf (float *x*) ;

Function	Arguments	Return Value
floorf	*x*: Numeric value to perform operation	Normal: The maximum integer no more than *x* When *x* is non-numeric or infinite: NaN When *x* = -0: +0 When the maximum integer no more than *x* cannot be expressed: *x*

EXPLANATION

- 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

FUNCTION

- fmodf finds the remainder of x/y.

HEADER

- math.h

FUNCTION PROTOTYPE

- float fmodf (float *x* , float *y*) ;

Function	Arguments	Return Value
fmodf	*x*: Numeric value to perform operation *y*: Numeric value to perform operation	Normal: Remainder of x/y When $x \neq \infty$ and $y = \pm\infty$: *x* When *x* is non-numeric or *y* is non-numeric, When *y* is $\pm$0, when *x* is $\pm\infty$: NaN

EXPLANATION

- 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 $\pm$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.

10.12 Diagnostic Functions

The following diagnostic functions are available.

- __assertfail

__assertfail

FUNCTION

- __assertfail supports assert macro.

HEADER

- assert.h

FUNCTION PROTOTYPE

- int __assertfail (char *__*msg* , char *__*cond* , char *__*file* , int __*line*) ;

Function	Arguments	Return Value
__assertfail	__*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

EXPLANATION

- A __assertfail function receives information from assert macro (refer to 10.2 (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.

10.13 Batch Files for Update of Startup Routine and Library Functions

The CC78K0R is provided with batch files for updating a part of the standard library functions and the startup routine. The batch files in the bat folder are shown in the table below.

Table 10-4 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 ROM 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 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 output port. When it is necessary to change this setting, change the defined value of EQU of PORT in putchar.asm and update the library using this batch file.
repputcs.bat	Updates the putchar function to SM+ for 78K0R-supporting. When it is necessary to check the output of the putchar function using the SM+ for 78K0R, 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/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/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.

10.13.1 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, an environment in which the RA78K0R assembler package Ver.1.00 or later operates is necessary. Before using the batch files, set the folder that contains the RA78K0R 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.

To use the batch files, move the current folder to the subfolder bat and execute each batch file. At this time, the following parameters are necessary.

Product type = chiptype (classification of target chip)

f1166a0 ... u PD78F1166_A0, etc.

The following is an illustration of how to use each batch file.

The batch file for:

(1) Startup routine

mkstup *chiptype*

<Example>

```
mkstup f1166a
```

(2) Firmware ROM routine update

reprom *chiptype*

<Example>

```
reprom f1166a0
```

(3) getchar function update

repgetc *chiptype*

<Example>

```
repgetc f1166a0
```

(4) putchar function update

repputc *chiptype*

<Example>

```
repputc f1166a0
```

(5) putchar function (SM78K0R-supporting) update

repputcs *chiptype*

<Example>

```
repputcs f1166a0
```

(6) setjmp/longjmp function update (with restore/save processing)

repselo *chiptype*

<Example>

```
repselo f1166a0
```

(7) setjmp/longjmp function update (without restore/save processing)

repselon *chiptype*

<Example>

```
repselon f1166a0
```

(8) Interrupt vector table update

repvect *chiptype*

<Example>

```
repvect f1166a0
```

CHAPTER 11 EXTENDED FUNCTIONS

This chapter describes the extended functions unique to the CC78K0R and not specified in the ANSI (American National Standards Institute) Standard for C.

The extended functions of the CC78K0R are used to generate codes for effective utilization of the target devices in the 78K0R. Not all of these extended functions are always effective. Therefore, it is recommended to use only the effective ones according to the user's purpose. For the effective use of the extended functions, refer to "CHAPTER 13 EFFECTIVE UTILIZATION OF COMPILER" along with this chapter.

C source programs created by using the extended functions of the CC78K0R utilize microcontroller-dependent functions. As regards portability to other microcontrollers, they are compatible at the C language level. For this reason, C source programs developed by using these extended functions are portable to other microcontrollers with easy-to-make modifications.

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

11.1 Macro Names

The CC78K0R has 2 types of macro names: those indicating the microcontroller names for target devices and those indicating device names (processor types). These macro names are specified according to the option at compile time to output object code for a specific target device or according to the processor type in the C source. In the example below, __K0R__ and __F1166A0_ are specified.

For details of these macro names, see "9.8 Compiler-Defined Macro Names".

<Example>

```
Compile option
        >CC78K0R -cf1166a0 prime.c ...

Specification of device type:
        #pragma pc ( f1166a0 )
```

11.2 Keywords

The CC78K0R is added with the following tokens as keywords to realize the extended function. These tokens cannot be used as labels nor variable names as well as ANSI-C keywords. All the keywords must be described in lowercase letters. A keyword containing any uppercase letter is not interpreted as such by the C compiler.

The following shows the list of keywords added to the CC78K0R. Of these keywords, ones not starting with "__" can be disabled by specifying the option (-za) that enables only ANSI-C language specifications (for the ANSI-C keywords, refer to "2.2 Keywords").

Table 11-1 List of Added Keywords

Keyword		Use
Always Available	Unavailable When -za Option Is Specified	
__callt	callt	callt/__callt functions
__callf	callf	callf/__callf functions
__sreg	sreg	sreg/__sreg variables
-	noauto	noauto functions
__leaf	norec	norec/__leaf functions
__boolean	boolean	boolean type/__boolean type variables
-	bit	bit type variables
__interrupt	-	Hardware interrupt
__interrupt_brk	-	Software interrupt
__banked, _non_banked		Bank Interface
__BANK0-15	-	Bank functions at constant addresses
__asm	-	ASM statements
__rtos_interrupt	-	Handler to allocate for RTOS
__pascal	-	Pascal function
__flash	-	Firmware ROM function
__flashf	-	__flashf function
__directmap	-	Absolute address allocation specification
__temp	-	Temporary variable
__near, __far	-	Memory allocation area specification
__mxcall	-	__mxcall function

(1) Functions

The keywords callt, __callt, __interrupt, __interrupt_brk, __rtos_interrupt, __flash, __flashf are attribute qualifiers. These keywords must be described before any function declaration.

The format of each attribute qualifier is shown below.

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

<Example>

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

Attribute qualifier specifications are limited to those listed below. callt and __callt are regarded as the same specifications. However, the qualifier added with "__" are enabled even when the -za option is specified.

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

Caution If callf, __callf, noauto, __pascal, __mxcall, norec, and __leaf are described, a warning is output and the description is ignored.

(2) Variables

- The same regulations apply to the sreg or __sreg specification as to the register in C language (refer to "11.5 How to use the saddr area (sreg/__sreg)" for details).
- The same regulations apply to the bit, boolean or __boolean specification as to the char or int type specifier in C language.
  However, these types can be specified only to the variables defined outside a function (external variables).
- The same regulations apply to the __directmap specification as to the type qualifier in C language (refer to 11.5 Absolute address allocation specification (__directmap) for details).
- The same regulations apply to the __near and __far specification as to the type qualifier in C language (refer to "11.5 near/far area specification" for details).

Caution If __temp is described, a warning is output and the description is ignored.

11.3 Memory

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

(1) Memory model

The following types of memory models are available.

Memory Model	Explaination
Small model (selected when the -ms option is specified)	Memory model that consists of a code portion of 64 KB (max.) and a data portion of 64 KB; 128 KB in total
Medium model (selected when the -mm option is specified)	Memory model that consists of a code portion of 1 MB (max.) and a data portion of 64 KB (max.)
Large model (selected when the -ml option is specified)	Memory model that consists of a code portion of 1 MB (max.) and a data portion of 1 MB (max.)

The data portion includes ROM data.

(2) Register bank

- The register bank is set to "RB0" at startup (set in the startup routine of the CC78K0R). The register bank 0 is made always used (unless the register bank is changed) by this setting.
- The specified register bank is set at the start of the interrupt function that has specified the change of the register bank.

(3) Memory space

The CC78K0R uses memory space as shown below.

Figure 11-1 Utilization of Memory Space

Address		Use		Size (bytes)
00	080 to 0BFH	CALLT table		64
FF	E20 to EB3H	sreg variables, boolean type variables		148
FF	EB4 to EC3H	Register variables		16
FF	EC4 to ED3H	Work		16
FF	ED4 to ED7H	Segment information		4
FF	ED8 to EDFH	Arguments of runtime library		8
FF	EE0 to EF7H	RB3 to RB1	Work registers[Note 1]	24
	EF8 to EFFH	RB0	Work registers	8
FF	F00 to FFFH	sfr variables		256
F0	000 to 7FFH	2nd sfr variables		Max 2,048[Note 2]

Note 1 Used when a register bank is specified.

Note 2 Varies depending on the device used.

11.4 #pragma Directive

The #pragma directive is one of the preprocessing directives supported by ANSI. The #pragma directive, depending on the character string to follow #pragma, instructs the compiler to translate in the method determined by the compiler.

If the compiler does not support the #pragma directive, the #pragma directive is ignored and compilation is continued. In the case that keywords are added depending on the directive, an error is output if the C source includes the keywords. In order to avoid this, either the keywords in the C source should be deleted or sorted by #ifdef directive.

The CC78K0R supports the following #pragma directives to realize the extended functions.

The keywords specified after #pragma can be described either in uppercase or lowercase letters.

For the extended functions using #pragma directives, refer to "11.5 How to Use Extended Functions".

Table 11-2 List of #pragma Directives

#pragma Directive	Applications
#pragma sfr	Describes SFR name in C -> "11.5 How to use the sfr area (sfr)"
#pragma vect #pragma interrupt	Describes interrupt processing in C -> "11.5 Interrupt functions (#pragma vect/#pragma interrupt)"
#pragma di #pragma ei	Describes DI/EI instructions in C -> "11.5 Interrupt functions (#pragma DI, #pragma EI)"
#pragma halt #pragma stop #pragma brk #pragma nop	Describes CPU control instructions in C -> "11.5 CPU control instruction (#pragma HALT/STOP/BRK/NOP)"
#pragma section	Changes compiler output section name and specify section location -> "11.5 Changing compiler output section name (#pragma section ...)"
#pragma name	Changes module name -> "11.5 Module name changing function (#pragma name)"
#pragma rot	Uses rotate function -> "11.5 Rotate function (#pragma rot)"
#pragma mul	Uses multiplication function -> 11.5 Multiplication function (#pragma mul)
#pragma div	Uses division function -> 11.5 Division function (#pragma div)
#pragma opc	Uses data insertion function -> 11.5 Data insertion function (#pragma opc)
#pragma rtos_interrupt	Uses interrupt handler for RX78K0R (real-time OS) -> 11.5 Interrupt handler for RTOS (#pragma rtos_interrupt ...)
#pragma rtos_task	Uses task function for RX78K0R (real-time OS) -> 11.5 Task function for RTOS (#pragma rtos_task)
#pragma ext_table	Specifies the first address of the flash area branch table -> 11.5 Flash area branch table (#pragma ext_table)

#pragma Directive	Applications
#pragma ext_func	Calls a function to the flash area from the boot area -> 11.5 Function of function call from boot area to flash area (#pragma ext_func)
#pragma inline	Expands the standard library functions memcpy and memset inline -> 11.5 Memory manipulation function (#pragma inline)

11.5 How to Use Extended Functions

The types of extended functions are given below.

- callt functions (callt/__callt)
- Register variables (register)
- How to use the saddr area (sreg/__sreg)
- How to use the sfr area (sfr)
- bit type variables, boolean type variables (bit/boolean/__boolean)
- ASM statements (#asm - #endasm/__asm)
- Kanji (2-byte character) (/* kanji */, // kanji)
- Interrupt functions (#pragma vect/#pragma interrupt)
- Interrupt function qualifier (__interrupt, __interrupt_brk)
- Interrupt functions (#pragma DI, #pragma EI)
- CPU control instruction (#pragma HALT/STOP/BRK/NOP)
- Bit field declaration
- Changing compiler output section name (#pragma section ...)
- Binary constant (Binary constant 0bxxx)
- Module name changing function (#pragma name)
- Rotate function (#pragma rot)
- Multiplication function (#pragma mul)
- Division function (#pragma div)
- BCD operation function (#pragma bcd)
- Data insertion function (#pragma opc)
- Interrupt handler for RTOS (#pragma rtos_interrupt ...)
- Interrupt handler qualifier for RTOS (__rtos_interrupt)
- Task function for RTOS (#pragma rtos_task)
- Flash area allocation method (-zf)
- Flash area branch table (#pragma ext_table)
- Function of function call from boot area to flash area (#pragma ext_func)
- Firmware ROM function (__flash)
- Method of int expansion limitation of argument/return value (-zb)
- Memory manipulation function (#pragma inline)
- Absolute address allocation specification (__directmap)

- near/far area specification
- Memory model specification

This section describes each of these extended functions in the following format:

FUNCTION

Outlines a function that can be implemented with the extended function.

EFFECT

Explains the effect brought about by the extended function.

USAGE

Explains how to use the extended function.

EXAMPLE

Indicates an application example of the extended function.

RESTRICTIONS

Explains restrictions if any on the use of the extended function.

EXPLANATION

Explains the above application example.

COMPATIBILITY

Explains the compatibility of a C source program developed by another C compiler when it is to be compiled with the CC78K0R.

callt functions (callt/__callt)

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
```

EXAMPLE

```
__callt void func1 ( void ) ;

__callt void func1 ( void ) {
                :
        /* function body */
                :
}
```

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 -ql4
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 ============          ============ ca2.c ============
__callt extern  int     tsub ( void) ;

void    main ( void )                   __callt int     tsub ( void )
{                                       {
        int     ret_val ;                       int val ;
        ret_val = tsub ( ) ;                    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
                  :
```

EXPLANATION

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

COMPATIBILITY

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R to another C compiler>

- #define must be used. For details, see "11.6 Modifications of C Source".

Register variables (register)

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, refer to "11.7 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
```

EXAMPLE

```
void    main ( void ) {
        register        unsigned char  c ;
                :
}
```

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 ( ) ;
}
```

[Example of register variable allocation to register HL and the saddr area]

The following labels are declared in the startup routine (refer to "APPENDIX A LIST OF LABELS FOR saddr AREA").

<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;  j = 1;
        movw    hl , #00H           ; The following codes are output in
                                    ; the middle of the function
        onew    ax
        movw    _@KREG00 , ax       ; j
; line  5 :     i += j;
        addw    ax , hl
        movw    hl , ax
; line  6 :
        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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

(a) Usage with sreg declaration

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 (refer to Figure 11-1).

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/short, 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 not made in the C source, the CC78K0R outputs the following codes. In this case, the ORG quasi-directive will not be output.

<Assembler source>

```
                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.

<Output object of compiler>

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

COMPATIBILITY

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R to another C compiler>

- Modifications are made by #define. For the details, refer to "11.6 Modifications of C Source". Thereby, sreg variables are handled as ordinary variables.

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

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	When *n* = 1: Variables of char and unsigned char types 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 (a).

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.

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

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	When *n* = 1: Variables of char and unsigned char types 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 (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 (a).

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)

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:

  (i) Comment

  (ii) 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.

<Output object of compiler>

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

COMPATIBILITY

<From another C compiler to the CC78K0R>

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

<From the CC78K0R 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, boolean type variables (bit/boolean/__boolean)

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 instruction
```

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).
- 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.

<Assembler source>

```
PUBLIC  _data1          ; Declaration
PUBLIC  _data2

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

The following codes are output in a function

<Output object of compiler>

```
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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R to another C compiler>

- #define must be used. For details, see "11.6 Modifications of C Source" (As a result of this, the bit or boolean type variables are handled as ordinary variables.).

ASM statements (#asm - #endasm/__asm)

FUNCTION

#asm - #endasm

- The assembler source program described by the user can be embedded in an assembler source file to be output by the CC78K0R by using the preprocessor directives #asm and #endasm.
- #asm and #endasm lines will not be output.

__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

#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
```

__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.
- 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

#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.

__asm

<C source>

```
int     a , b ;

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

<Assembler source>

```
@@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)

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

(a) Set "SJIS", "EUC" or "NONE". (Also describe it in files such as autoexec.bat, if necessary.)

(b) Specification of SJIS, EUC or NONE is not case-sensitive.

(c) 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 ) {
; line          3 : {
@@CODEL CSEG
_main :
; line          4 :             /* Comment */
; line          5 : }
```

EXPLANATION

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

COMPATIBILITY

<From another C compiler to the CC78K0R>

- If there is kanji in the the area that comment cannot be described (the area other than “/* ... */”, or “// newline”), 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.

<From the CC78K0R 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)

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):

  (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

  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, refer to the user's manual of the target device used). For the software interrupt BRK, describe BRK_I.
- To describe #pragma PC (processor type), describe this #pragma directive after that. The following items can be described before this #pragma directive:

  (i) Comments

  (ii) Preprocessor directive which does neither define nor refer to a variable or a function

<In the case of the static model>

```
#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.

  Refer to 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 CC78K0R 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.

  (i) If two or more interrupts occur

  (ii) If two or more interrupts that use the same BANK are included in the interrupt that has occurred

  (iii) 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

[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
```

[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 ( void ) ;
}
```

<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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

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.

  <For non-maskable/maskable interrupt function>

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

  <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, refer to 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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

FUNCTIONS

- 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:

  (i) Comment

  (ii) Other #pragma directives

  (iii) 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
```

[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
        ; Post-processing
        ret
```

COMPATIBILITY

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

FUNCTION

- The following codes are output to the object to create an object file:

  (1) Instruction for HALT operation (HALT)

  (2) Instruction for STOP operation (STOP)

  (3) BRK instruction

  (4) 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:

  (i) Comment

  (ii) Other #pragma directive

  (iii) 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:

  (1) HALT ( ) ;

  (2) STOP ( ) ;

  (3) BRK ( ) ;

  (4) 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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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

(a) Extension of type specifier

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).

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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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.

(b) Allocation direction of bit field

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.

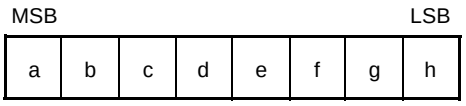
EXAMPLE 1

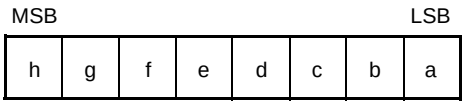
<Bit field declaration>

```
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



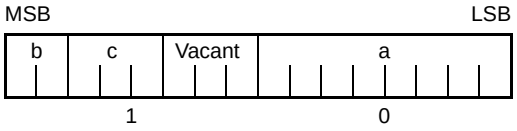


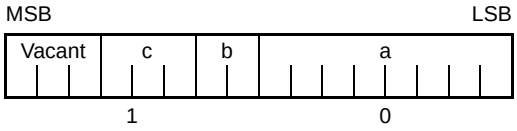
EXAMPLE 2

<Bit field declaration>

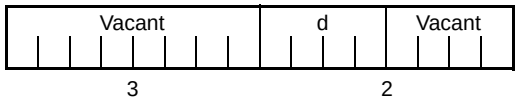
```
struct  t {
        char            a ;
        unsigned char   b : 2 ;
        unsigned char   c : 3 ;
        unsigned char   d : 4 ;
        int             e ;
        unsigned char   f : 5 ;
        unsigned char   g : 6 ;
        unsigned char   h : 2 ;
        unsigned int    i : 2 ;
} ;
```

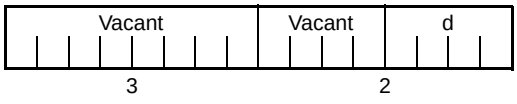
Bit field allocated from the MSB side when the -rb option is specified	Bit field allocated from the LSB side when the -rb option is not specified

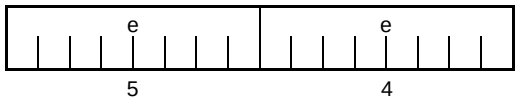


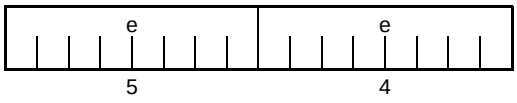


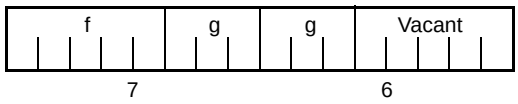
Member a of char type is allocated to the first byte unit. Members b and c are allocated to subsequent byte units, starting from the second byte unit. If a byte unit does not have enough space to hold the type char member, that member will be allocated to the following byte unit. In this case, if there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

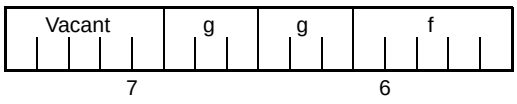


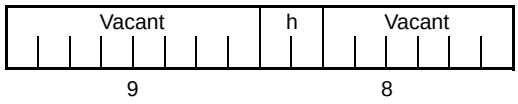


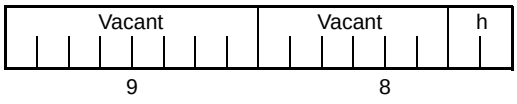




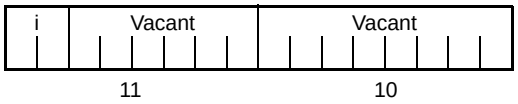


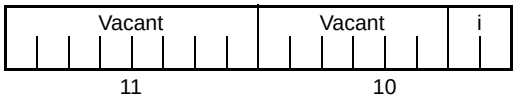






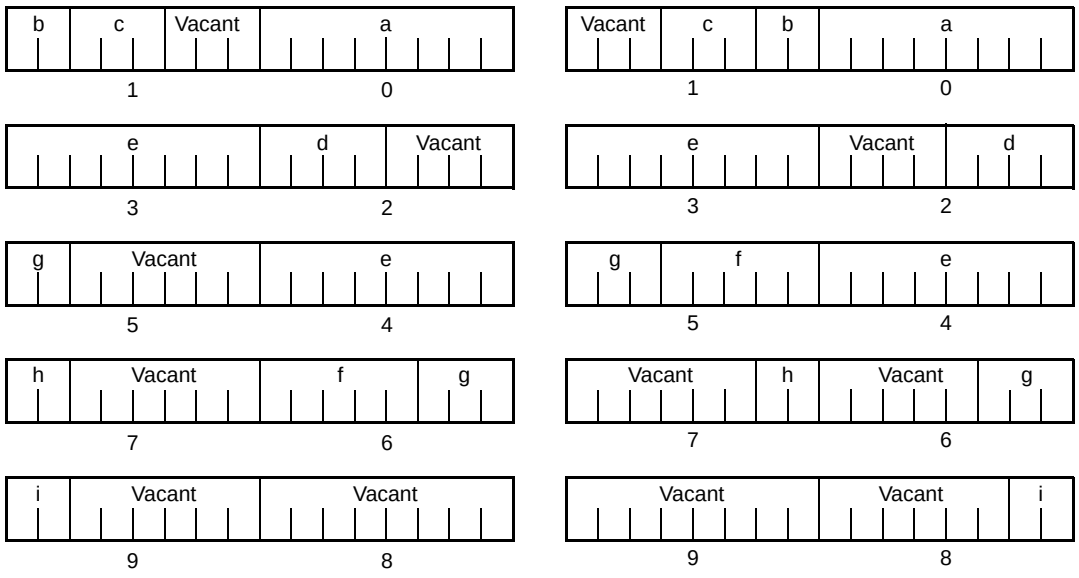
Since member g is a bit field of type unsigned int, it can be allocated across byte boundaries. Since h is a bit field of type unsigned char, it is not allocated in the same byte unit as the g bit field of type unsigned int, but is allocated in the next byte unit.





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.



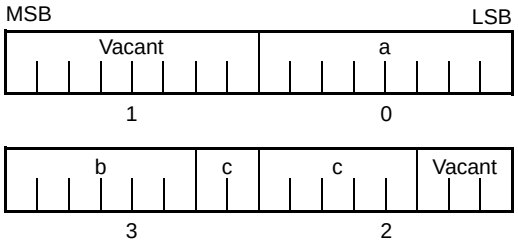
Remark The numbers below the allocation diagrams indicate the byte offset values from the beginning of the structure.

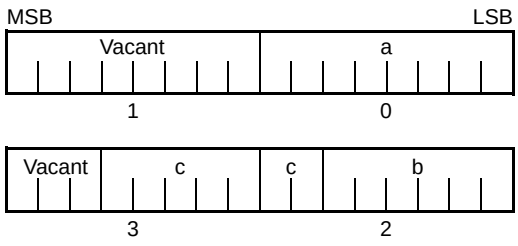
EXAMPLE 3

<Bit field declaration>

```
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 ;
} ;
```

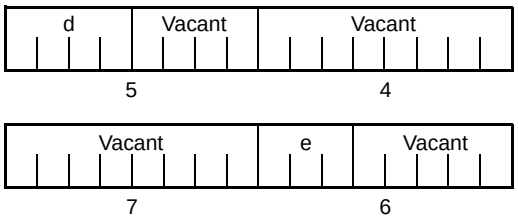
Bit field allocated from the MSB side when the -rb option is specified	Bit field allocated from the LSB side when the -rb option is not specified

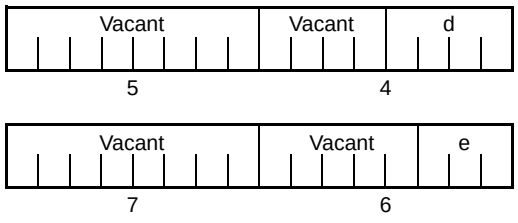




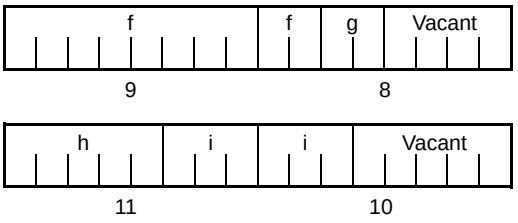
Since b and c are bit fields of type unsigned int, they are allocated from the next word unit.

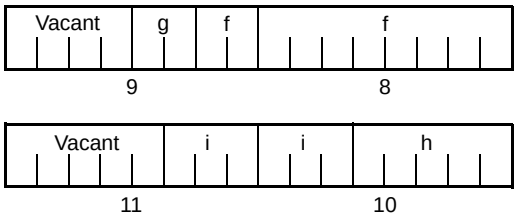
Since d is also a bit field of type unsigned int, it is allocated from the next word unit.





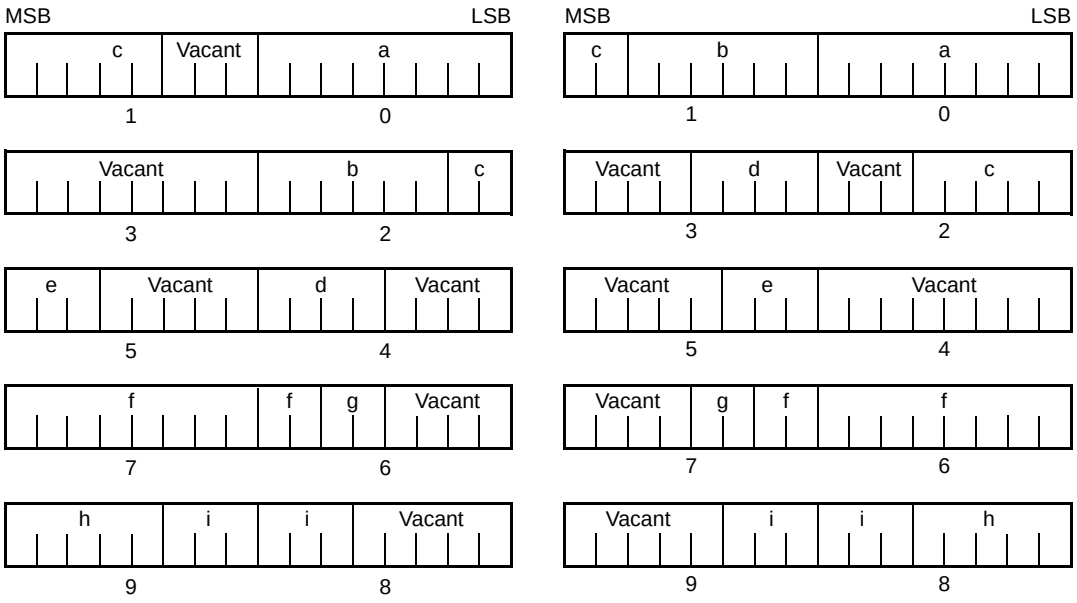
Since e is a bit field of type unsigned char, it is allocated to the next byte unit.





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

<From another C compiler to the CC78K0R>

- The source program need not be modified.

<From the CC78K0R to another C compiler>

- The 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 ...)

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, refer to "APPENDIX B 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, refer to the RA78K0R Assembler Package Operation User's Manual.
- 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.
  Describe this #pragma directive after #pragma PC (processor type).
  The following items can be described before this #pragma directive:

  (i) Comment

  (ii) 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 start-address]
```

- 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.

[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)
```

[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.

<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 (refer to "CODING Error EXAMPLE 1").
- #include statement cannot be described in a file which executes this #pragma instruction following the C text description. If described, it causes an error (refer to "CODING Error EXAMPLE 2").
- If #include statement follows the C text, this #pragma instruction cannot be described after this description. If described, it causes an error (refer to "CODING Error EXAMPLE 3").

EXAMPLE 1

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>

```
CC1     CSEG    AT      2400H
_main :
        ; Preprocessing
        ; Function body
        ; Post-processing
        ret
```

EXAMPLE 2

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 "//".

```
#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
// 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
```

EXAMPLE 3

```
#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.

CODING ERROR EXAMPLE 1

```
a1.h
        #pragma section @@DATA  ??DATA1         // File containing only the
                                                // #pragma section.

a2.h
        extern  int     func1( void ) ;
        #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"
```

CODING ERROR EXAMPLE 2

```
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.
```

CODING ERROR EXAMPLE 3

```
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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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.

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, and from 0x0 to 0xffeff for other sections.

CAUTION

- 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.
- 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.
- When the -zf option has been specified, each section name is changed so that the second "@" is replaced with "E".

Note ROMization-related section name

```
@@R_INIT, @@R_INIS, @@RLINIT, @@INITL, @@INIT, @@INIS
```

The startup routine to be used when a section related to ROMization is changed, and an example of changing the termination routine are described later.

[Examples of Changing startup Routine in Connection with Changing Section Name Related to ROMization]

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.

(a) Changing startup routine cstartx.asm

(i) 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
  :
```

(ii) 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

(iii) 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
```

(b) Changing termination routine rom.asm

(i) 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
  :
```

(ii) 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 (Binary constant 0bxxx)

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

<From another C compiler to the CC78K0R>

- Modifications are not needed.

<From the CC78K0R 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)

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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

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
```

[List of functions for rotate]

(a) unsigned char rorb (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;

Rotates *x* to right for *y* times.

(b) unsigned char rolb (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;

Rotates *x* to left for *y* times.

(c) unsigned int rorw (*x* , *y*) ;
unsigned int *x* ;
unsigned char *y* ;

Rotates *x* to right for *y* times.

(d) 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.
  (i) Comments
  (ii) Other #pragma directives
  (iii) 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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

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.

```
mulu
```

[List of multiplication function]

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.
  (i) Comments
  (ii) Other #pragma directives
  (iii) 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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

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 CC78K0 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
```

[List of division function]

(a) unsigned int divuw (*x* , *y*) ;
unsigned int *x* ;
unsigned char *y* ;

Performs unsigned division of *x* and *y* and returns the quotient.

(b) 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.
  (i) Comments
  (ii) Other #pragma directives
  (iii) 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 in line, 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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

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, btobcd, bcdtow, wtobcd, and bbcd 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. Refer to [List of functions for BCD operation], later in this chapter for more information.

```
adbcdb, sbbcdb, adbcdbe, sbbcdbe, adbcdw, sbbcdw, adbcdwe,
sbbcdwe, bcdtob, btobcde, bcdtow, wtobcd, btobcd
```

- 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.

  (i) Comments

  (ii) Other #pragma directives

  (iii) Preprocessing directives that do not generate definitions/references 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 assembler source>

```
        mov     a , !_a
        add     a , !_b
        add     a , !BCDADJ
        mov     !_c , a
        mov     a , !_b
        sub     a , !_a
        sub     a , !BCDADJ
        mov     !_c , a
```

[List of functions for BCD operation]

(a) unsigned char adbcdb (x , y) ;

unsigned char x ;

unsigned char y ;

Decimal addition is carried out by the BCD adjustment instruction.

(b) unsigned char sbbcdb (x , y) ;

unsigned char x ;

unsigned char y ;

Decimal subtraction is carried out by the BCD adjustment instruction.

(c) unsigned int adbcdbe (x , y) ;

unsigned char x ;

unsigned char y ;

Decimal addition is carried out by the BCD adjustment instruction (with result expansion).

(d) 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.

(e) unsigned int adbcdw (x , y) ;

unsigned int x ;

unsigned int y ;

Decimal addition is carried out by the BCD adjustment instruction.

(f) unsigned int sbbcdw (x , y) ;

unsigned int x ;

unsigned int y ;

Decimal subtraction is carried out by the BCD adjustment instruction.

(g) unsigned long adbcdwe (x , y) ;

unsigned int x ;

unsigned int y ;

Decimal addition is carried out by the BCD adjustment instruction (with result expansion).

(h) 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.

(i) unsigned char bcdtob (x) .

unsigned char x ;

Values in decimal number are converted to binary number values.

(j) unsigned int btobcde (x) ;

unsigned char x ;

Values in binary number are converted to decimal number values.

(k) unsigned int bcdtow (x) ;

unsigned int x ;

Values in decimal number are converted to binary number values.

(l) 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.

(m) unsigned char btobcd (x) ;

unsigned char x ;

Values in decimal number are converted to those in binary number.

However, the overflow is discarded.

COMPATIBILITY

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

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.

  [List of data insertion functions]

  (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.

  (i) Comments

  (ii) Other #pragma directives

  (iii) 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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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 ...)

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.
- This #pragma directive is described at the start of the C source.
  When #pragma PC (type) is described, main #pragma directive is described after #pragma pc.
  The following can be described before the #pragma directive.

  (i) Comments

  (ii) Preprocessing directives which do not generate definition/reference of variables and definition/ reference of functions

```
#pragma∆rtos_interrupt[∆interrupt-request-name∆function-name]
```

- 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 6 :              int     a [ 3 ] ;
; line 7 :              a [ 0 ] = 1;
                onew    ax
                movw    [ hl ] , ax     ; a
; line 8 :              func () ;
                call    !!_func
; line 9 :      }
                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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R to another C compiler>

- Changes can be made by #define (For the details, refer to "11.6 Modifications of C Source"). 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)

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.

```
#pragma∆rtos_task[∆task-function-name]
```

- The #pragma directives are described at the start of the C source.
  However, the followings can be described before the #pragma directive.

  (i) Comments

  (ii) 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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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)

Caution This function enables the flash memory rewriting function of devices.

FUNCTIONS

- 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 compilation.

RESTRICTION

- Use startup routines or library for the flash area.

Flash area branch table (#pragma ext_table)

Caution This function enables the flash memory rewriting function of devices.

FUNCTIONS

- 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.

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

- The following #pragma instruction specifies the first address of the flash area branch table.

```
#pragma ext_table       branch-table-first-address
```

  Describe the #pragma instruction at the beginning of C source.

- The following items can be described before the #pragma instruction:

  (i) Comments

  (ii) #pragma instructions other than #pragma ext_func, #pragma vect with -zf specification, #pragma interrupt, or #pragma rtos_interrupt.

  (iii) Instructions not to generate the definition/reference of variables or functions among the preprocess instructions.

RESTRICTIONS

- The branch table is located at the first address of the flash area.
- If #pragma ext_table does not exist before #pragma ext_func, #pragma vect with -zf specification, #pragma interrupt, or #pragma rtos_interrupt, an error will occur.
- The first address of the branch table is assumed to be 80H to 0FF80H0xc0 to 0xff00.
- It is necessary to reconfigure the library for interrupt vectors (_@vect00 to _@vect7e) in accordance with the specified first address of the branch table. The default is 2000H in the interrupt vector library. To specify the value other than 2000H, reconfigure the library as shown below.

  (i) Change the place of H in ITBLTOP EQU 2000H of vect.inc in the \Program Files\NEC Electronics Tools\CC78K0R\Vx.xx\src\cc78k0r\src folder to the specified address.

(ii) Run \Program Files\NEC Electronics Tools\CC78K0R\Vx.xx\src\cc78k0r\bat\repvect.bat in command prompt, and update library by assembly. Copy the updated library \Program Files\NEC Electronics Tools\CC78K0R\Vx.xx\src\cc78k0r\lib to \Program Files\NEC Electronics Tools\CC78K0R\Vx.xx\lib78k0r to be used for link.

Caution The above folder may differ depending on the installation method.

COMPATIBILITY

<From another C compiler to the CC78K0R>

- If #pragma ext_table is not used, correction is not necessary.
- To specify the first address of the flash area branch table, change the address in accordance with USAGE above.

<From the CC78K0R to another C compiler>

- Delete the #pragma ext_table instruction or divide it by #ifdef.
- To specify the first address of the flash area branch table, the following change is required.

EXAMPLE

[To generate a branch table after the address 2000H and place the interrupt function:]

<C source>

```
#pragma ext_table       0x2000
#pragma interrupt       INTP0   intp

void    intp ( void ) {
}
```

(a) To place the interrupt function to the boot area (no -zf specified)

<Output code>

```
                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.

(b) To place the interrupt function in the flash area (-zf specified)

<Output code>

```
                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 2009H 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)

Caution This function enables the flash memory rewriting function of devices.

FUNCTIONS

- 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.

(i) Comments

(ii) 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).
- #pragma ext_table does not exist before #pragma ext_func, it results in an error.
- For the same function with a different ID value and a different function with the same ID value, an error will occur. (a) and (b) below are errors.

(a) #pragma ext_func f1 3
#pragma ext_func f1 4

(b) #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.

COMPATIBILITY

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R 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 ext_table       0x2000
#pragma ext_func        f1      3
#pragma ext_func        f2      4

extern  void    f1 ( void ) ;
extern  void    f2 ( void ) ;

void    func ( void )
{
        f1 ( ) ;
        f2 ( ) ;
}

(2) Flash area side

#pragma ext_table       0x2000
#pragma ext_func        f1      3
#pragma ext_func        f2      4

void    f1 ( void ) {
}

void    f2 ( void ) {
}
```

Remark 1 #pragma ext_func f1 3 means that the branch destination to function f1 is located in branch table address 2000H + 4 * 64 + 4 * 3.

Remark 2 #pragma ext_func f2 4 means that the branch destination to function f2 is located in branch table address 2000H + 4 * 64 + 4 * 4.

Remark 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) Boot area side (without -zf specification)

@@CODEL CSEG
_func :
        call    !0210CH
        call    !02110H
        ret

(2) Flash area side (with -zf specification)

@ECODEL CSEG
_f1 :
        ret
_f2 :
        ret

@EXT03  CSEG    AT      0210CH
        br      !!_f1
        br      !!_f2
```

Firmware ROM function (__flash)

Caution This function enables the flash memory rewriting function of devices.

FUNCTIONS

- This calls a firmware ROM function which self-writes to the flash memory via the interface library positioned between the firmware ROM function and the C language function.
- In the interface library call interface, the first argument is passed to the register and the second and subsequent arguments are transferred to the stack. The first argument's register is as follows.

1- or 2-byte data:	AX
4-byte data:	AX (low-order), BC (high-order)

- It is necessary that the interface library be set to the return the values in the following registers according to the size of return values.

1- or 2-byte data:	BC
near pinter:	BC
4-byte data, far pointer:	BC (low-order), DE (high-order)

EFFECT

- The operations related to the firmware ROM function can be described at the C source level.

USAGE

- During interface library prototype declaration, __flash attributes are added to the top.

RESTRICTIONS

- Function calls by a function pointer are not supported.
- When a function with __flash is defined, it results in an error.

COMPATIBILITY

<From another C compiler to the CC78K0R>

- If the reserved word __flash is not used, corrections are not necessary.
- If you desire to change the firmware ROM function, use the USAGE above.

<From the CC78K0R to another C compiler>

- Possible using #define (refer to "11.6 Modifications of C Source").
- In a CPU with a firmware ROM function or substitute function, it is necessary for the user to create an exclusive library to access that area.

Method of int expansion limitation of argument/return value (-zb)

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 compilation.

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 ;
}
```

[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 :     unsigned char   func1（unsigned char x , unsigned char y）
; line 9 :      {
_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

<From another C compiler to the CC78K0R>

- 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.

<From the CC78K0R to another C compiler>

- No modification is needed.

Memory manipulation function (#pragma inline)

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.

  (i) Comments

  (ii) Other #pragma directives

  (iii) 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

<From another C compiler to the CC78K0R>

- Modification is not needed if the memory manipulation function is not used.
- When changing the memory manipulation function, use the method above.

<From the CC78K0R to another C compiler>

- The #pragma inline directive should be deleted or delimited using #ifdef.

Absolute address allocation specification (__directmap)

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 to 0xfffff	0x00000 to 0xfffff
Secured Area Range	0xffd00 to 0xffeff	0xffd00 to 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 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>

```
        PUBLIC  _main
_c      EQU     0FFE00H   ; Addresses for variables declared by __directmap
are defined by EQU
_d      EQU     0FFE20H
_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
specified in saddr area
; line 12 :
        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

<From another C compiler to the CC78K0R>

- No modification is necessary if the keyword __directmap is not used.
- To change to the __directmap variable, modify according to the description method above.

<From the CC78K0R to another C compiler>

- Compatibility can be attained using #define (refer to "11.6 Modifications of C Source" 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

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.

  (1) "0xf" is added to the higher bytes of the variable pointer (0 is exceptional and zero-extended).

  (2) 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.

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

```
__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.

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

<From another C compiler to the CC78K0R>

- It is not necessary to modify the code if reserved word __near or __far is not used.

<From the CC78K0R 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.

CAUTION

- 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

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: 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
```

<Output object of compiler: 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
```

<Output object of compiler: 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 (64 K - 1) bytes each, per load module file (Note: 64 KB 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").

11.6 Modifications of C Source

By using the extended functions of the CC78K0R, efficient object generation can be realized. However, these extended functions are intended to cope with the 78K0R. So, to use them for other devices, the C source may need to be modified.

Here, how to make the C source portable from another C compiler to the CC78K0R and vice versa is explained.

<From another C compiler to the CC78K0R>

- #pragma[Note]

  If the other C compiler supports the #pragma preprocessor 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.

<From the CC78K0R to another C compiler>

- Because the CC78K0R 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.

EXAMPLE

(1) To invalidate a keyword (Same applies to callf, sreg, and norec, etc.)

```
#ifndef __K0R__
#define callt           /* Makes callt as ordinary function */
#endif
```

(2) To change from one type to another

```
#ifndef __K0R__
#define bit     char    /* Changes bit type to char type variable */
#endif
```

11.7 Function Call Interface

The following will be explained about the interface between functions at function call.

(1) Return value (common in all the functions)

(2) Ordinary function call interface

(a) Passing arguments

(b) Location at which arguments and automatic variables are stored

11.7.1 Return value

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.

Type	Location of Storing
1-byte integer	BC
2-byte integer	
4-byte integer	BC (Lower), DE (Upper)
Pointer (__near attribute)	BC
Pointer (__far attribute)	BC (Lower), DE (Upper)
Structure, union (Small model, Medium model)	BC
Structure, union (Large model)	BC (Lower), DE (Upper)
1 bit	CY (carry flag)
Floating-point number (float type)	BC (Lower), DE (Upper)
Floating-point number (double type)	BC (Lower), DE (Upper)

11.7.2 Ordinary function call interface

(1) Passing arguments

- When a function is called, the second argument and later are passed to the function definition side via a 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 11-3 Details of Type Modification (Change from int and short Type to char Type)

Type	Location of Storing
1-byte data[Note] 2-byte data[Note]	AX
3-byte data[Note] 4-byte data[Note]	AX, BC
Floating-point number	AX, BC
Others	Passed via stack

Note 1- to 4-byte data include structure, union, and pointer.

(2) Location at which arguments and automatic variables are stored

- 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 "APPENDIX A LIST OF LABELS FOR saddr AREA" 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.

[Example]

<C source 1>

```
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 ;
}
```

[When the -qr optoon is specified]

<Output Code>

```
_main :
; line  4 :     func0 ( 0x1234 , 0x5678 ) ;
        movw    ax ,  #05678H   ; 22136
        push    ax                      ; The second argument and later are
                                        ; passed via a stack
        movw    ax , #01234H    ; 4660  ; The first argument that is passed
                                        ; via a register
        call    !!_func0                ; Function call
        pop     ax                      ; Releases the stack in which data
                                        ; is stored upon function call
; line  5 : }
        ret
; line  6 :
; line  7 : void func0 ( register int p1 , int p2 ) {
_func0 :
        push    hl
        movw    de , _@KREG14
        push    de                      ; Saves the saddr area for a
                                        ; register variable
        movw    de , _@KREG12
        push    de                      ; Saves the saddr area for a
                                        ; register variable
        movw    _@KREG14 , ax           ; Allocate register argument p1 to
                                        ; saddr
        push    ax                      ; Reserves area for the automatic
                                        ; 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     ; Automatic variable r
; line 11 :     a = p1 ;
        movw    ax , _@KREG14   ; p1    ; Argument p1
        movw    [ hl ] , ax     ; a     ; Automatic variable a
; line 12 : }
        pop     ax                      ; Reserves area for the automatic
                                        ; variable a
        pop     ax
        movw    _@KREG12 , ax           ; Restores the saddr area for a
                                        ; register argument
        pop     ax
        movw    _@KREG14 , ax           ; Restores the saddr area for a
                                        ; register argument
        pop     hl
        ret
```

<C source 2>

```
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 ;
}
```

[When the -qr option is specified]

<Output Code>

```
_main :
; line  4 :     func0 ( 0x1234 , 0x5678 ) ;
        movw    ax , #05678H    ; 22136
        push    ax                      ; The second argument and later are
                                        ; passed via a stack
        movw    ax , #01234H    ; 4660  ; The first argument that is passed
                                        ; via a register
        call    !!_func1                ; Function call
        pop     ax                      ; Releases the stack in which data
                                        ; is stored upon function call
; line  5 : }
        ret
; line  6 :
; line  7 : void func0 ( int p1 , register int p2 ) {
_func0 :
        push    hl
        push    ax                      ; Loads the first argument p1 on the
                                        ; stack
        movw    de , _@KREG14
        push    de                      ; Saves the saddr area for register
                                        ; variables
        movw    de , _@KREG12
        push    de                      ; Saves the saddr area for register
                                        ; variables
        movw    ax , [ sp + 12 ]
        movw    _@KREG12 , ax           ; Allocates the argument p2 to saddr
        push    ax                      ; Reserves area for the automatic
                                        ; 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     ; Automatic variable r
; line 11 :     a = p1 ;
        movw    ax , [ hl + 6 ] ; p1    ; Argument p1
        movw    [ hl ] , ax     ; a     ; Automatic variable a
; line 12 : }
        pop     ax                      ; Releases area for the automatic
                                        ; variable a
        pop     ax
        movw    _@KREG12 , ax           ; Restores the saddr area for
                                        ; register arguments
        pop     ax
        movw    _@KREG14 , ax           ; Restores the saddr area for
                                        ; register arguments
        pop     ax                      ; Releases area for the first
                                        ; argument p1
        pop     hl
        ret
```

CHAPTER 12 REFERENCING THE 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.

How to interface with another language by using the RA78K0R Assembler Package and the CC78K0R is described in this order:

- Calling Assembly Language Routines from C Language
- Calling C Language Routines from Assembly Language
- Referencing variables defined in the C language
- Referencing variables defined in the assembly language from the C language
- Cautions

12.1 Accessing Arguments/Automatic Variables

For details on assignment of argument and automatic variables, refer to "11.7.2 Ordinary function call interface".

Register HL is used as a base pointer for accessing arguments and automatic variables stored in a stack.

12.2 Storing Return Values

Refer to “11.7.1 Return value“.

12.3 Calling Assembly Language Routines from C Language

This section shows examples of default procedures.

Calling an assembly language routine from the C language is described as follows.

- C language function calling procedure
- Saving data from the assembly language routine and returning

12.3.1 C language function calling procedure

This is a C language program example that calls an assembly language routine.

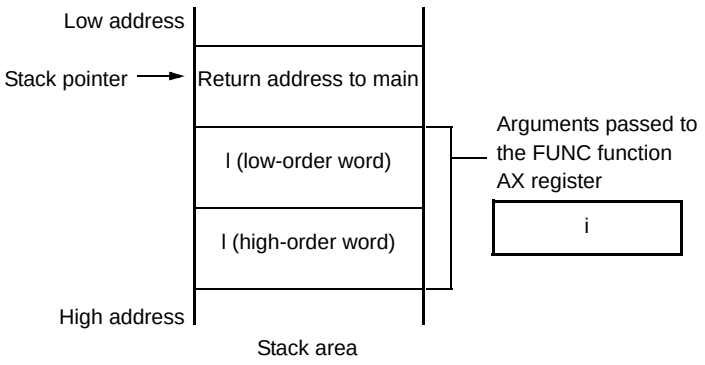
```
extern  int     FUNC ( int , long ) ;   /* Function prototype */

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.

(i) 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.

(ii) 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.



12.3.2 Saving data from the assembly language routine and returning

The following processing are performed in the FUNC function called from 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.

Next, an example of an assembly language program is explained.

```
$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, work register

A label with "_" prefixed to the function name described in the C source is described. Base pointers and work registers are saved with the same name as function names described inside the C source.

After the label is described, 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 variable. is not required.

(2) Copying to base pointer (HL) of stack pointer (SP)

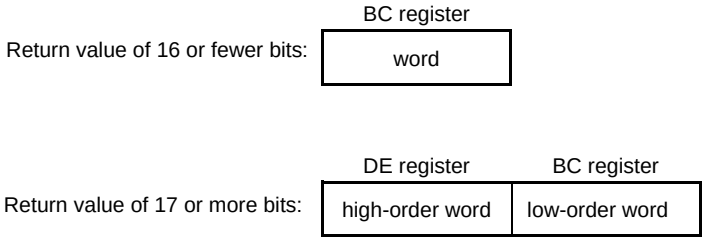
The stack pointer (SP) changes due to "PUSH, POP" inside functions. Therefore, the stack pointer is copied to register "HL" and used as the base pointer of arguments.

(3) Basic processing of FUNC function

After processings (1) and (2) are performed, the basic processing of called functions is performed.

(4) Setting the return value

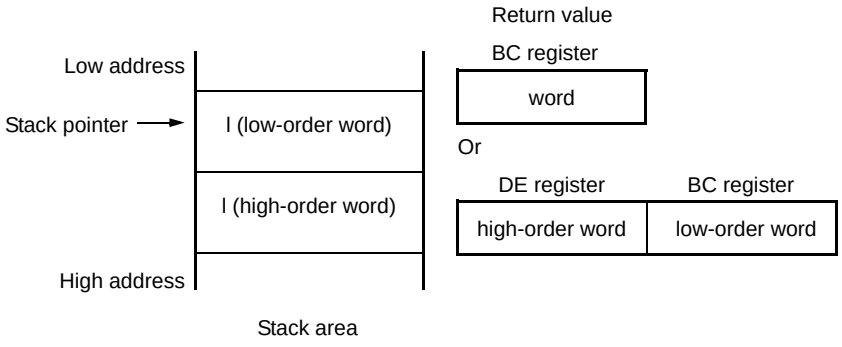
If there is a return value, it is set in the "BC" and "DE" registers. If there is no return value, setting is unnecessary.



(5) Restoring the registers

Restore the saved base pointer and work register.

(6) Returning to the main function



12.4 Calling C Language Routines from Assembly Language

12.4.1 Calling the C language function from an assembly language program

The procedure for calling a function written in the C language from an assembly language routine is:

(1) Save the C work registers (AX, BC, and DE).

(2) Place the arguments on the stack.

(3) Call the C language function.

(4) Increment the value of the stack pointer (SP) by the number of bytes of arguments.

(5) 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.

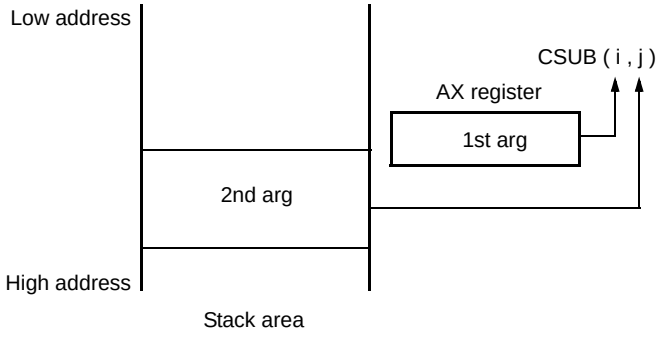
Save or restore the registers before or after an argument pass code.

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.



(3) Calling a C language function

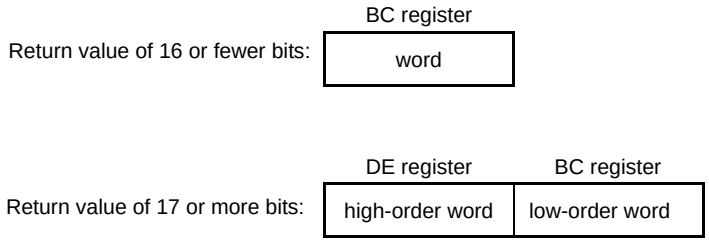
A CALL instruction calls a C language function. If the C language function is a "callt" function, the callt instruction performs the call, and if a "callf" function, the callf instruction performs it.

(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.



12.5 Referencing Variables Defined in Other Languages

12.5.1 Referencing variables defined in the 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.

<C language program example>

```
extern  void    subf ( void ) ;

char    c = 0 ;
int     i = 0 ;

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

The following occurs in the RA78K0R assembler.

```
$PROCESSOR ( F1166A0 )

        PUBLIC  _subf
        EXTRN   _c
        EXTRN   _i

@@CODE  CSEG
_subf :
        MOV     !_c , #04H
        MOVW    AX , #07H
        MOVW    !_i , AX
        RET
        END
```

12.5.2 Referencing variables defined in the assembly language from the C language

Variables defined in assembly language are referenced from the C language in this way.

<C language program example>

```
extern  char    c ;
extern  int     i ;

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

The following occurs in the RA78K0R assembler.

```
        NAME    ASMSUB

        PUBLIC  _i
        PUBLIC  _c

ABC     DSEG    BASEP
_i :    DW      0
_c :    DB      0

        END
```

12.6 Cautions

(1) "_" (underscore)

The CC78K0R adds an underscore "_" (ASCII code "5FH") to external definitions and reference names of the object modules to be output.

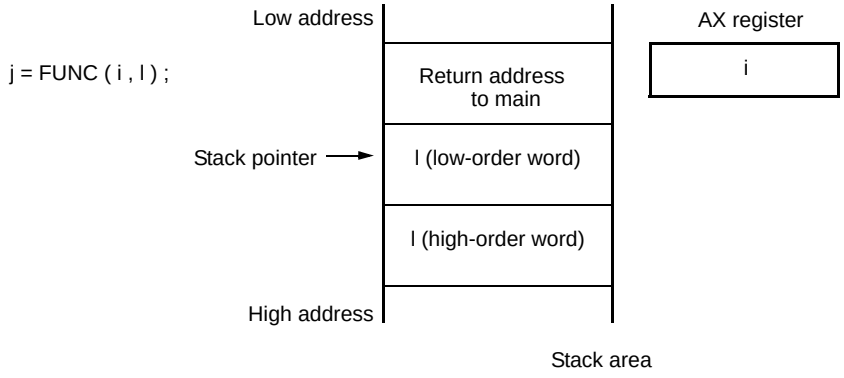
In the next C program example, "j = FUNC(i, l);" is taken as "a reference to the external name _FUNC".

```
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 RA78K0R.

(2) 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.



CHAPTER 13 EFFECTIVE UTILIZATION OF COMPILER

This chapter introduces how to effectively use the CC78K0R.

13.1 Efficient Coding

When developing 78K0R microcomputer-applied products, efficient object generation may be realized with the CC78K0R by utilizing the saddr area, callt area of the device.

- Use external variables
  - if (saddr area is usable)
    - sreg/__sreg variables are used/ compiler option (-rd) is used
- Use 1-bit data
  - if (saddr is usable)
    - bit/boolean/ __boolean type variables are used
- Function definition
  - if (function to be called several times)
    - if (callt area is usable)
      - Use as __callt/callt function (effective for reducing code size)
  - if (automatic variables are used &&saddr area is usable)

(1) Using external variable

When defining an external variable, specify the external variable to be defined as a sreg/__sreg variable if the saddr area can be used. Instructions to sreg/__sreg variables are shorter in code length than instructions to memory. This helps shorten object code and improve program execution speed. (The same can be also performed by specifying the -rd option, instead of using the sreg variable.)

```
Definition-of-sreg/__sreg-variable: extern sreg int     variable-name ;
                                    extern __sreg int   variable-name ;
```

Remark Refer to "11.5 How to use the saddr area (sreg/__sreg)".

(2) 1-bit data

A data object which only uses 1-bit data should be declared as a bit type variable (or boolean/__boolean type variable). A bit manipulation instruction will be generated for an operation on bit/boolean/__boolean type variable. Because saddr area is used as well as sreg variable, the codes can be shortened and the execution speed can be improved.

```
Declaration-of-bit/boolean-type-variable:  bit         variable-name ;
                                           boolean     variable-name ;
                                           __boolean   variable-name ;
```

Remark Refer to "11.5 bit type variables, boolean type variables (bit/boolean/__boolean)".

(3) Function definitions

If the callt table can be used for functions to be called frequently, such functions should be defined as callt functions.

The callt functions can be called faster than ordinary function calls with shorter codes because the callt functions are called using the callt/callf area of the device.

```
Definition-of-callt-function:   callt   int     tsub ( )  {
                                    :
                                }
```

Remark Refer to "11.5 callt functions (callt/__callt)".

In addition to the use of the saddr area, the objects that do not need the modification of the C source by compiling with the optimization option can be generated. For the effect of each -q suboption, refer to the CC78K0R C Compiler Operation User's Manual.

(4) Using extended description

- Function definition

  - if (automatic variables are used && saddr area is usable)
    - register declaration
  - if (internal static variables are used) && (saddr area is usable)
    - __sreg declaration

- Functions which use automatic variables

  If the saddr area can be used for a function that does not use automatic variables, declare the function with the register storage class specifier. By this register declaration, the object declared as register will be allocated to a register.

  A program using registers operates faster than that using memory and object code can be shortened as well.

Remark Refer to "11.5 Register variables (register)" about definition of register variable (register int i; ...).

- Functions which use internal static variables

  If the saddr area can be used for a function that uses internal static variables, declare the function with __sreg or specify the -rs option. In the same way as with sreg variables, the object code can be shortened and the execution speed can be improved.

  Remark Refer to "11.5 How to use the saddr area (sreg/__sreg)".

In addition, the code efficiency and the execution speed can be improved in the following method.

- Use of SFR name (or SFR bit name).

  #pragma sfr

- Use of __sreg declaration for bit fields which consist only of 1-bit members (unsigned char type can be used for members).

```
__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 ;
```

- Use of the register bank change for interrupt processing.

  #pragma interrupt INTP0 inter RB1

- Use of multiplication and division embedded function.

  #pragma mul

  #pragma div

- Description of only the modules whose speed needs to be improved in the assembly language.

APPENDIX A LIST OF LABELS FOR saddr AREA

In the CC78K0R, the saddr area is referenced by the following label names. Therefore, the label names in the C source program and in assembler source program that have the same names as the following cannot be used.

(a) 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.

(b) For work

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

(c) For storing segment information

Label Name	Address
_@SEGAX	0FFED4H
_@SEGBC	0FFED5H
_@SEGDE	0FFED6H
_@SEGHL	0FFED7H

(d) Arguments of runtime library

Label Name	Address
_@RTARG0	0FFED8H
_@RTARG1	0FFED9H
_@RTARG2	0FFEDAH
_@RTARG3	0FFEDBH
_@RTARG4	0FFEDCH
_@RTARG5	0FFEDDH
_@RTARG6	0FFEDEH
_@RTARG7	0FFEDFH

APPENDIX B LIST OF SEGMENT NAMES

This chapter explains all the segments that the compiler outputs and their locations.

(1) and (2) show the option and re-allocation attributes used in the table.

(1) CSEG re-allocation attribute

- 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).

(2) DSEG re-allocation attribute

- 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).

B.1 List of Segment Names

B.1.1 Program area and data area

Section Name	Segment Type	Re-allocation 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	- Segment for const variable (allocated to near area) - References table for switch-case statement - Unnamed character string constants - Initial value data of auto variables
@@CNSTL	CSEG	PAGE64KP	Segment for const variable (allocated to far area)
@@R_INIT	CSEG	UNIT64KP	Segment for near initialization data (with initial value)
@@RLINIT	CSEG	UNIT64KP	Segment for far initialization data (with initial value)
@@R_INIS	CSEG	UNIT64KP	Segment for initialization 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]
@@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)
@@BITS	BSEG		Segment for boolean-type and bit-type variables

Note The value of *nn* and *mm* changes depending on the interrupt types.

B.1.2 Flash memory area

Section Name	Segment Type	Re-allocation 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	Segment for const variable (allocated to near area)
@ECNSTL	CSEG	PAGE64KP	Segment for const variable (allocated to far area)
@ER_INIT	CSEG	UNIT64KP	Segment for near initialization data (with initial value)
@ERLINIT	CSEG	UNIT64KP	Segment for far initialization data (with initial value)
@ER_INIS	CSEG	UNIT64KP	Segment for initialization data (sreg variable with initial value)
@EVECT*nn*	CSEG	AT *mmmm*H	Segment for vector table[Note 1]
@EXT*xx*	CSEG	AT *yyyy*H	Segment for flash area branch table[Note 2]
@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

Note 1 The value of *nn* and *mm* changes depending on the interrupt types.

Note 2 The values of xx and *yyyy* vary depending on the ID of the flash area function.

B.2 Location of Segment

Table B-1 Location of Segment

Segment Type	Destination of Allocation (Default)
CSEG	ROM
BSEG	saddr area of RAM
DSEG	RAM

B.3 Example of C Source

```
#pragma INTERRUPT       INTP0   inter   rb1     /* Interrupt vector */

void    main ( void );          /* Interrupt 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 ;            /* External variable with initial value */
int     i_data ;                /* External variable without initial value */
__sreg  int     sr_inis = 3 ;   /* sreg variable with initial value */
__sreg  int     sr_dats ;       /* sreg variable without initial value */

void    main ( void )           /* Function definition */
{
        int     i ;
        i = 100 ;
}

void    inter ( void )          /* Interrupt function definition */
{
}

callt   void    f_clt ( void )  /* callt function definition */
{
}
```

B.4 Example of Output Assembler Module

Quasi-directives and instruction sets in an assembler source vary depending on the device.

Refer to the RA78K0R Assembler Package Operation User’s Manual for details.

[When specifying 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 , 0330H , 00H , 020H , 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 boolean-type and bit-type
                                    ; variable
_b_bit      DBIT

@@CNST      CSEG  MIRRORP           ; Segment for const variable
_i_cnst :   DW    01H           ; 1

@@R_INIT    CSEG  UNIT64KP          ; Segment for initialization data
                                    ; (External variable with initial value)
            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
                                    ; (sreg variable with initial value)
            DW    03H           ; 3

@@INIS      DSEG  SADDRP            ; Segment for tentative data area
                                    ; (sreg variable with initial value)
_sr_inis :  DS    ( 2 )

@@DATS      DSEG  SADDRP            ; Segment for tentative data area
                                    ; (sreg variable without initial value)
_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 vector */
```

```
; line  2 :
; line  3 : void    main ( void ) ;             /* Interrupt function */
                                                /* prototype declaration */
; line  4 : const   int     i_cnst = 1 ;        /* const variable */
; line  5 : callt   void    f_clt ( void ) ;    /* callt function prototype */
; line  6 : boolean b_bit ;                     /* boolean-type variable */
; line  7 : long    l_init = 2 ;                /* External variable */
                                                /* with initial value */
; line  8 : int     i_data ;                    /* External variable */
                                                /* without initial value */
; line  9 : sreg    int     sr_inis = 3 ;       /* sreg variable with */
                                                /* initial value */
; line 10 : sreg    int     sr_dats ;           /* sreg variable without */
                                                /* initial value */
; line 11 :
; line 12 : void    main ( )                    /* Function definition */
; line 13 : {

@@CODE   CSEG  BASE                  ; Segment for code portion
_main :
         push  hl
; line 14 :         int     i ;
; line 15 :         i = 100 ;
         movw  hl , #064H         ; 100
; line 16 : }
         pop   hl
         ret
; line 17 :
; line 18 : void    inter ( )        /* Interrupt function definition */
; line 19 : {

@@BASE   CSEG  BASE                 ; Segment for callt and interrupt function
_inter :
; line 20 : }
         reti
; line 21 :
; line 22 : callt   void f_clt ( )   /* callt function definition */
; line 23 : {
_f_clt:
; line 24 : }
         ret
@@VECT06 CSEG  AT      0006H         ; Segment for vector table
_@vect06 :
         DW    _inter
         END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

[When specifying 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 , 0330H , 00H , 020H , 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 boolean-type and bit-type
                                    ; variable
_b_bit      DBIT

@@CNST      CSEG  MIRRORP           ; Segment for const variable
_i_cnst :   DW    01H           ; 1

@@R_INIT    CSEG  UNIT64KP          ; Segment for initialization data
                                    ; (External variable with initial value)
            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
                                    ; (sreg variable with initial value)
            DW    03H           ; 3

@@INIS      DSEG  SADDRP            ; Segment for tentative data area
                                    ; (sreg variable with initial value)
_sr_inis :  DS    ( 2 )

@@DATS      DSEG  SADDRP            ; Segment for tentative data area
                                    ; (sreg variable without initial value)
_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 ) ;    /* callt function prototype */
                                                /* declaration */
; line  6 : boolean b_bit ;                      /* boolean-type variable */
; line  7 : long    l_init = 2 ;    /* External variable with initial value */
; line  8 : int     i_data ;    /* External variable without initial value */
; line  9 : __sreg  int     sr_inis = 3 ;       /* sreg variable with */
                                                /* initial value */
; line 10 : __sreg  int     sr_dats ;           /* sreg variable without */
                                                /* initial value */
; line 11 :
; line 12 : void    main ( )                    /* Function definition */
; line 13 : {

@@CODEL  CSEG                                   ; Segment for code portion
_main :
         push  hl
; line 14 :         int     i ;
; line 15 :         i = 100 ;
         movw  hl , #064H          ; 100
; line 16 : }
         pop   hl
         ret
; line 17 :
; line 18 : void    inter ( )       /* Interrupt function definition */
; line 19 : {

@@BASE   CSEG  BASE                ; Segment for callt and interrupt function
_inter :
; line 20 : }
         reti
; line 21 :
; line 22 : callt   void f_clt ( )             /* callt function definition */
; line 23 : {
_f_clt:
; line 24 : }
         ret
@@VECT06 CSEG  AT      0006H                   ; Segment for vector table
_@vect06 :
         DW    _inter
         END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

[When specifying 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 , 0330H , 00H , 020H , 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 boolean-type and bit-type
                                        ; variable
_b_bit      DBIT

@@R_INIS    CSEG  UNIT64KP              ; Segment for initialization data
                                        ; (External variable with initial value)
            DW    03H             ; 3

@@INIS      DSEG  SADDRP                ; Segment for tentative data area
                                        ; (sreg variable with initial value)
_sr_inis :  DS    ( 2 )

@@DATS      DSEG  SADDRP                ; Segment for tentative data area
                                        ; (sreg variable without initial value)
_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 ) ;    /* callt function prototype */
; line  6 : __boolean       b_bit ;             /* boolean-type variable */
; line  7 : long    l_init = 2 ;  /* External variable with initial value */
; line  8 : int     i_data ;    /* External variable without initial value */
; line  9 : __sreg  int     sr_inis = 3 ;       /* sreg variable with */
                                                /* initial value */
; line 10 : __sreg  int     sr_dats ;           /* sreg variable without */
                                                /* initial value */
; line 11 :
; line 12 : void    main ( )                    /* Function definition */
; line 13 : {

@@CODEL  CSEG                                   ; Segment for code portion
_main :
         push  hl
; line 14 :         int     i ;
; line 15 :         i = 100 ;
         movw  hl , #064H           ; 100
; line 16 : }
         pop   hl
         ret
; line 17 :
; line 18 : void    inter ( )       /* Interrupt function definition */
; line 19 : {

@@BASE   CSEG  BASE                 ; Segment for callt and interrupt function
_inter :
; line 20 : }
         reti
; line 21 :
; line 22 : callt   void f_clt ( )              /* callt function definition*/
; line 23 : {
_f_clt:
; line 24 : }
         ret
@@VECT06 CSEG  AT      0006H                   ; Segment for vector table
_@vect06 :
         DW    _inter
         END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

APPENDIX C LIST OF RUNTIME LIBRARIES

The table below shows the runtime library list.

These operational instructions are called in the format where @@, etc. are attached at the beginning of the function name. However, cstart, cstarte, cprep, and cdisp are called in the format with _@ attached to the top.

No library supports are available for operations not in Table C-1.

The compiler executes in-line development.

long addition and subtraction, and/or/xor and shift may be developed in-line.

Table C-1 Runtime Libraries

Classification	Function Name	Function
Increment	lsinc	Increments signed long
	luinc	Increments unsigned long
	finc	Increments float
Decrement	lsdec	Decrements signed long
	ludec	Decrements unsigned long
	fdec	Decrements float
Sign reverse	lsrev	Reverses the sign of signed long
	lurev	Reverses the sign of unsigned long
	frev	Reverses the sign of float
1's complement	lscom	Obtains 1's complement of signed long
	lucom	Obtains 1's complement of unsigned long
Logical NOT	lsnot	Negates signed long
	lunot	Negates unsigned long
	fnot	Negates float
Multiply	csmul	Performs multiplication between signed char data
	cumul	Performs multiplication between unsigned char data
	iumul	Performs multiplication between signed int, unsigned int data
	lsmul	Performs multiplication between signed long data
	lumul	Performs multiplication between unsigned long data
	fmul	Performs multiplication between float data

Table C-1 Runtime Libraries

Classification	Function Name	Function
Divide	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
Remainder	csrem	Obtains remainder after division between signed char data
	curem	Obtains remainder after division between unsigned char data
	isrem	Obtains remainder after division between signed int data
	iurem	Obtains remainder after division between unsigned int data
	lsrem	Obtains remainder after division between signed long data
	lurem	Obtains remainder after division between unsigned long data
Add	lsadd	Performs addition between signed long data
	luadd	Performs addition between unsigned long data
	fadd	Performs addition between float data
Subtract	lssub	Performs subtraction between signed long data
	lusub	Performs subtraction between unsigned long data
	fsub	Performs subtraction between float data
Shift left	lslsh	Shifts singed long data to the left
	lulsh	Shifts unsigned long data to the left
Shift right	lsrsh	Shifts signed long data to the right
	lursh	Shifts unsigned long data to the right
Compare	cscmp	Compares signed char data
	iscmp	Compares signed int data
	lscmp	Compares signed long data
	lucmp	Compares unsigned long data
	fcmp	Compares float data
Bit AND	lsband	Performs an AND operation between signed long data
	luband	Performs an AND operation between unsigned long data
Bit OR	lsbor	Performs an OR operation between signed long data
	lubor	Performs an OR operation between unsigned long data
Bit XOR	lsbxor	Performs an XOR operation between signed long data
	lubxor	Performs an XOR operation between unsigned long data

Table C-1 Runtime Libraries

Classification	Function Name	Function
Conversion from floating-point number	ftols	Converts from float to signed long
	ftolu	Converts from float to unsigned long
Conversion to floating-point number	lstof	Converts from signed long to float
	lutof	Converts from unsigned long to float
Conversion from bit	btol	Converts from bit to long

Startup routine	cstart	Startup module
		- After an area (4 * 32 bytes) where a function that will be registered is reserved with the atexit function, sets the beginning label name to _@FNCTBL.
		- Reserve a break area (32 bytes), sets the beginning label name to _@MEMTOP, and then sets the next label name of the area to _@MEMBTM.
		- Define the segment in the reset vector table as follows, and set the beginning address of the startup module.
		`@@VECT00 CSEG AT 0000H`
		`DW _@cstart`
		- Sets a mirror area.
		- Set the register bank to RB0.
		- Set 0 to the variable _errno to which the error number is input.
		- Set the variable _@FNCENT, to which the number of functions registered by the atexit function is input, to 0.
		- Set the address of _@MEMTOP to the variable _@BRKADR as the initial break value.
		- 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 execute 0 clear of external data without an initial value.
		- Call the main function (user program)
		- Call the exit function by parameter 0.

Pre- and post-processing of function	cprep	Pre-processing of function
	cdisp	Post-processing of function
	cprep2	Pre-processing of function (including the saddr area for register variables)
	cdisp2	Post-processing of function (including the saddr area for register variables)
	cprep3	Pre-processing of function (including the saddr area for register variables)
	cdisp3	Post-processing of function (including the saddr area for register variables)
	hdwinit	Performs initialization processing of peripheral devices (sfr) immediately after CPU reset.

Table C-1 Runtime Libraries

Classification	Function Name	Function
BCD-type conversion	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

Table C-1 Runtime Libraries

Classification	Function Name	Function
Auxiliary	mulu	mulu instruction-compatible
	mulue	mulu instruction-compatible
	divuw	divuw instruction-compatible
	divuwe	divuw instruction-compatible
	swjmp2	2 bytes branch table for switch statement
	swjmp3	3 bytes branch table for switch statement
	swjmpr	Relative branch table for switch statement
	swjmpr2	Relative branch table for switch statement (There is compression.)
	cmpa1	For replacing the fixed-type instruction pattern
	cmpax1	
	ctoi	
	incde	
	decde	
	inchl	
	dechl	
	dellab	
	dell03	
	della4	
	delsab	
	dels03	
	hlllab	
	hlll03	
	hllla4	
	hllsab	
	hlls03	
	apinch	
	apdech	
	incwhl	
	decwhl	
	swap4	
	tableh	
	uctoi	
	rt03pu	
	rt47pu	

Table C-1 Runtime Libraries

Classification	Function Name	Function
Auxiliary	rt03po	For replacing the fixed-type instruction pattern
	rt47po	
	crac	
	prab	
	prad	
	lr04	
	lr40	
	lx04	
	inda	
	ifda	
	inad	
	ifad	
	pnda	
	pfda	
	pnad	
	pfad	
	lnd0	
	lnd4	
	lfd0	
	lfd4	
	ln0d	
	lf0d	
	cfdap1	
	cfdap2	
	ifdap1	
	pradp1	

Table C-1 Runtime Libraries

<table>
<tr><th>Classification</th><th>Function Name</th><th>Function</th></tr>
<tr><td>Auxiliary</td><td>VWXYZZ</td><td>Library for data transfer
- It follows the following naming conventions.

<table>
<tr><td>V</td><td>c: char
i: int
p: far pointer
l: long</td></tr>
<tr><td>W</td><td>n: near
f: far
r: Transfer between registers
x: Register exchange</td></tr>
<tr><td>X</td><td>Forwarding origin
a: ax
b: bc
d: de (Indirect reference)
h: hl (Indirect reference)
c: cs
0: _@RTARG0-3
4: _@RTARG45,ax</td></tr>
<tr><td>Y</td><td>Forwarding destination
a: ax
b: bc
d: de (Indirect reference)
h: hl (Indirect reference)
c: cs
0: _@RTARG0-3
4: _@RTARG45,ax</td></tr>
<tr><td>ZZ</td><td>If there are two or more kinds, I follow a number</td></tr>
</table>
</td></tr>
</table>

APPENDIX D LIST OF LIBRARY STACK CONSUMPTION

The table below shows the number of stacks consumed from the standard libraries.

Table D-1 List of Standard Library Stack Consumption

Classification	Function Name	Shared by Small Model and Medium Model	Shared by Large Model
ctype.h	isalnum	0	0
	isalpha	0	0
	iscntrl	0	0
	isdigit	0	0
	isgraph	0	0
	islower	0	0
	isprint	0	0
	ispunct	0	0
	isspace	0	0
	isupper	0	0
	isxdigit	0	0
	tolower	0	0
	toupper	0	0
	isascii	0	0
	toascii	0	0
	_tolower	0	0
	_toupper	0	0
	tolow	0	0
	toup	0	0
setjmp.h	setjmp	4	4
	longjmp	2	2
stdarg.h	va_arg	0	0
	va_start	0	0
	va_starttop	0	0
	va_end	0	0

Table D-1 List of Standard Library Stack Consumption

Classification	Function Name	Shared by Small Model and Medium Model	Shared by Large Model
stdio.h	sprintf	52 (102)[Note 1]	52 (112)[Note 1]
	sscanf	290 (330)[Note 1]	290 (338)[Note 1]
	printf	60 (100)[Note 1]	64 (104)[Note 1]
	scanf	302 (328)[Note 1]	306 (336)[Note 1]
	vprintf	60 (100)[Note 1]	66 (110)[Note 1]
	vsprintf	52 (100)[Note 1]	52 (110)[Note 1]
	getchar	0	0
	gets	8	14
	putchar	0	0
	puts	6	10
	__putc	4	4

Table D-1 List of Standard Library Stack Consumption

Classification	Function Name	Shared by Small Model and Medium Model	Shared by Large Model
stdlib.h	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 2]	6 + n[Note 2]
	abs	0	0
	div	6	6
	labs	0	0
	ldiv	16	16
	brk	0	0
	sbrk	2	2
	atof	18	18
	strtod	18 (30)[Note 6]	20 (34)[Note 6]
	itoa	10	10
	ltoa	16	16
	ultoa	16	16
	rand	18 (14)[Note 3]	18 (14)[Note 3]
	srand	0	0
	bsearch	40+ n[Note 4]	44 + n[Note 4]
	qsort	16 + n[Note 5]	18 + n[Note 5]
	strbrk	0	0
	strsbrk	2	2
	stritoa	10	10
	strltoa	16	16
	strultoa	16	16

Table D-1 List of Standard Library Stack Consumption

Classification	Function Name	Shared by Small Model and Medium Model	Shared by Large Model
string.h	memcpy	4	8
	memmove	4	6
	strcpy	2	6
	strncpy	4	10
	strcat	2	6
	strncat	4	8
	memcmp	2	4
	strcmp	2	2
	strncmp	2	2
	memchr	2	4
	strchr	4	2
	strcspn	4	4
	strpbrk	4	6
	strrchr	4	6
	strspn	4	6
	strstr	4	8
	strtok	4	4
	memset	4	6
	strerror	0	0
	strlen	0	0
	strcoll	2	2
	strxfrm	4	6

Table D-1 List of Standard Library Stack Consumption

Classification	Function Name	Shared by Small Model and Medium Model	Shared by Large Model
math.h	acos	34	34
	asin	34	34
	atan	34	34
	atan2	38	38
	cos	32	32
	sin	32	32
	tan	38	38
	cosh	38	38
	sinh	38	38
	tanh	44	44
	exp	34	34
	frexp	8 (20)Note 6	12 (24)Note 6
	ldexp	6 (20)Note 6	6 (22)Note 6
	log	34	34
	log10	34	34
	modf	8 (20)Note 6	12 (24)Note 6
	pow	38	38
	sqrt	26	26
	ceil	6 (18)Note 6	6 (20)Note 6
	fabs	4	4
	floor	6 (18)Note 6	6 (20)Note 6
	fmod	10 (22)Note 6	10 (24)Note 6
	matherr	0	0
	acosf	34	34
	asinf	34	34
	atanf	34	34
	atan2f	38	38
	cosf	32	32
	sinf	32	32
	tanf	38	38
	coshf	38	38
	sinhf	38	38

Table D-1 List of Standard Library Stack Consumption

Classification	Function Name	Shared by Small Model and Medium Model	Shared by Large Model
math.h	tanhf	44	44
	expf	34	34
	frexpf	8 (20)Note 6	12 (24)Note 6
	ldexpf	6 (20)Note 6	6 (22)Note 6
	logf	34	34
	log10f	34	34
	modff	8 (20)Note 6	12 (24)Note 6
	powf	38	38
	sqrtf	26	26
	ceilf	6 (18)Note 6	6 (20)Note 6
	fabsf	4	4
	floorf	6 (18)Note 6	6 (20)Note 6
	fmodf	10 (22)Note 6	10 (24)Note 6
assert.h	__assertfail	72 (112)Note 7	82 (122)Note 7

Note 1 Values in parentheses are for when the version that supports floating-point numbers is used.

Note 2 n is the total stack consumption among external functions registered by the atexit function.

Note 3 Values in the parentheses are for when a multiplier is used.

Note 4 n is the stack consumption of external functions called from bsearch.

Note 5 n is (X + stack consumption of external functions called from qsort) x (1 + number of times recursive calls occurred).

When using a library shared by small model and medium model: X = 38

When using a library shared by large model: X = 40

Note 6 Values in parentheses are for when an operation exception occurs.

Note 7 Values in parentheses are for when the printf version that supports floating-point numbers is used.

The table below shows the number of stacks consumed from the runtime libraries.

Table D-2 List of Runtime Library Stack Consumption

Classification	Function Name	Stack Consumption
Increment	lsinc	0
	luinc	0
	finc	20
Decrement	lsdec	0
	ludec	0
	fdec	20
Sign reverse	lsrev	2
	lurev	2
	frev	4
1's complement	lscom	0
	lucom	0
Logical NOT	lsnot	0
	lunot	0
	fnot	4
Multiply	csmul	0
	cumul	0
	iumul	4 (2)Note 2
	lsmul	8 (4)Note 2
	lumul	8 (4)Note 2
	fmul	12 (28, 30)Note 1
Divide	csdiv	8
	cudiv	2
	isdiv	10
	iudiv	4
	lsdiv	12
	ludiv	6
	fdiv	12 (28, 30) Note 1
Remainder	csrem	8
	curem	2
	isrem	12
	iurem	6
	lsrem	12
	lurem	6

Table D-2 List of Runtime Library Stack Consumption

Classification	Function Name	Stack Consumption
Add	lsadd	0
	luadd	0
	fadd	12 (28, 30)[Note 1]
Subtract	lssub	2
	lusub	2
	fsub	12 (28, 30)[Note 1]
Shift left	lslsh	4
	lulsh	4
Shift right	lsrsh	4
	lursh	4
Compare	cscmp	0
	iscmp	0
	lscmp	2
	lucmp	2
	fcmp	8 (26, 28)[Note 1]
Bit AND	lsband	0
	luband	0
Bit OR	lsbor	0
	lubor	0
Bit XOR	lsbxor	0
	lubxor	0
Conversion from floating-point number	ftols	10
	ftolu	10
Conversion to floating-point number	lstof	10
	lutof	10
Conversion from bit	btol	0
Startup routine	cstart	4

Table D-2 List of Runtime Library Stack Consumption

Classification	Function Name	Stack Consumption
Pre- and post-processing of function	cprep	2 + n[Note 3]
	cdisp	0
	cprep2	Size of + base pointer + first argument + register variable + automatic variable
	cdisp2	0
	cprep3	Size of + base pointer + first argument + register variable + automatic variable
	cdisp3	0
	hdwinit	0

Table D-2 List of Runtime Library Stack Consumption

Classification	Function Name	Stack Consumption
Auxiliary	mulu	0
	mulue	0
	divuw	6
	divuwe	6
	cmpa1	0
	cmpax1	0
	ctoi	0
	incde	0
	decde	0
	inchl	0
	dechl	0
	dellab	0
	dell03	0
	della4	0
	delsab	0
	dels03	0
	hlllab	0
	hlll03	0
	hllla4	0
	hllsab	0
	hlls03	0
	apinch	0
	apdech	0
	incwhl	0
	decwhl	0
	swap4	0
	tableh	0
	uctoi	0

Note 1 Values in parentheses are for when an operation exception occurs (when the matherr function included with the compiler is used).

(A, B)

A: When using a library shared by small model and medium model

B: When using a library shared by large model

Note 2 Values in the parentheses are for when a multiplier is used.

Note 3 n is the size of the automatic variable to be secured.

APPENDIX E LIST OF MAXIMUM INTERRUPT DISABLED TIME FOR LIBRARIES

Time during which an interrupt is disabled is provided for libraries for which a multiplier is used in order that the contents of the operation are not destroyed during an interrupt.

The table below shows the maximum interrupt disabled time for libraries for which a multiplier is used.

No periods during which an interrupt is disabled are provided for libraries for which a multiplier is not used.

Table E-1 Maximum Interrupt Disabled Time (Number of Clocks) for Libraries

Classification	Function Name	Maximum Interrupt Disabled Time	Remark
Multiplication	@@iumul	12	Performs multiplication between signed int, unsigned int data
	@@lsmul	24	Performs multiplication between signed long data
	@@lumul	24	Performs multiplication between unsigned long data
stdlib.h	rand	24	@@lumul is used
	qsort	12	@@iumul is used

INDEX

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