User's Manual

NEC

CC78K4 Ver.2.30 or Later

C Compiler

Language

Target Devices:
78K/IV Series

Document No. U15556EJ1V0UM00 (1st edition)
Date Published November 2001 N CP(K)

[MEMO]

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

PC/AT is a trademark of International Business Machines Corporation.

UNIX is a registered trademark licensed by X/Open Company Limited in the US and other countries.

SPARCstation is a trademark of SPARC International, Inc.

HP9000 series 700 is a trademark of Hewlett-Packard Company.

- **The information in this document is current as of July, 2001. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC's data sheets or data books, etc., for the most up-to-date specifications of NEC semiconductor products. Not all products and/or types are available in every country. Please check with an NEC 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 prior written consent of NEC. NEC assumes no responsibility for any errors that may appear in this document.
- NEC 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 semiconductor 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 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 customer's equipment shall be done under the full responsibility of customer. NEC assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC endeavours to enhance the quality, reliability and safety of NEC semiconductor 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 semiconductor products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment, and anti-failure features.
- NEC semiconductor products are classified into the following three quality grades:
  "Standard", "Special" and "Specific". The "Specific" quality grade applies only to semiconductor products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of a semiconductor product depend on its quality grade, as indicated below. Customers must check the quality grade of each semiconductor 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 semiconductor products is "Standard" unless otherwise expressly specified in NEC's data sheets or data books, etc. If customers wish to use NEC semiconductor products in applications not intended by NEC, they must contact an NEC sales representative in advance to determine NEC's willingness to support a given application.

  (Note)

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

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

M8E 00.4

Regional Information

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

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

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

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

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

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

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

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

NEC Electronics (France) S.A.
Velizy-Villacoublay, France
Tel: 01-3067-5800
Fax: 01-3067-5899

NEC Electronics (France) S.A.
Madrid Office
Madrid, Spain
Tel: 091-504-2787
Fax: 091-504-2860

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

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

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

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

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

NEC do Brasil S.A.
Electron Devices Division
Guarulhos-SP, Brasil
Tel: 11-6462-6810
Fax: 11-6462-6829

J01.2

INTRODUCTION

The **CC78K4 C Compiler** (hereafter 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, 78K/IV Series application products can be developed.

The **CC78K4 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 **CC78K4 C Compiler Operation User's Manual (U15557E)**.

For the architecture of 78K/IV Series, refer to the user's manual of each product of 78K/IV Series.

[Target Devices]

Software for the 78K/IV Series microcontrollers can be developed with this C compiler.
Note that the device file (sold separately) corresponding to the target device is necessary.

[Target 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 reader need not necessarily have knowledge of C compilers or C language. Discussions in this manual assume that 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 constituent 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 priority 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 PREPROCESSING DIRECTIVES

Details the types of preprocessing directives and how to use each preprocessing 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 provided to make the most of the target device.

CHAPTER 12 REFERENCING BETWEEN C AND ASSEMBLER

Describes the method of linking a C source program with a program written in assembly language.

CHAPTER 13 EFFECTIVE UTILIZATION

Outlines how to effectively use this C compiler.

APPENDIXES A through **E**

Contain a list of labels for **saddr** area, a list of segment names, a list of runtime libraries, a list of library stack consumption, and an 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 where in the manual details can be referenced.
- For those who are familiar with C compilers or C language:
  The language specifications of this C compiler conform to **ANSI Standard C**. Therefore, you may start from **CHAPTER 11,** which 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 78K/IV Series, if necessary.

[Related Documents]

Document Name	Document No.
CC78K4 C Compiler Operation User's Manual	U15557E

[Reference]

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

[Terms]

RTOS = **78K/IV Series Real-time OS RX78K/IV**

[Conventions]

The following conventions 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

LIST OF TABLES (1/2)

LIST OF TABLES (2/2)

CHAPTER 1 GENERAL

The CC78K4 C Compiler is a language processing program that converts a source program written in the C language for the 78K/IV Series or ANSI-C into machine language. By the CC78K4 C compiler, object files or assembler source files for the 78K/IV Series 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 human being (programmer) 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 programmer 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 a glance, increasing the likelihood of 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 programmer to write a program without regard to the architecture of the computer.

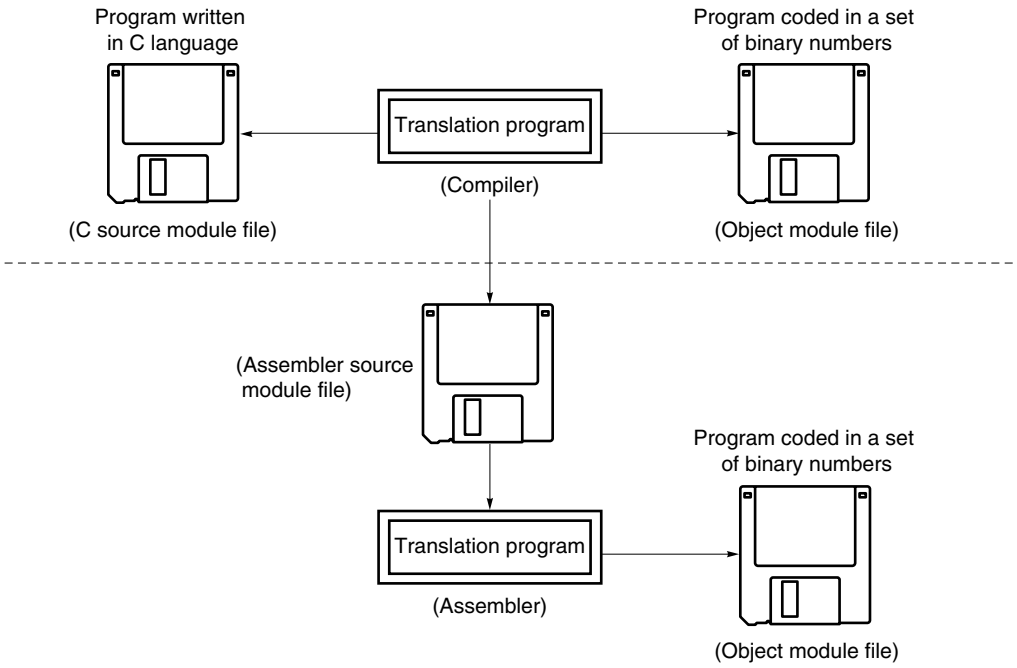
Compared with assembly language programs, it can be said that programs written in C have an easy-to-understand logic structure.

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

C is characterized by its ease of understanding by human 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 into the corresponding machine language instructions. A program that translates the C language into machine language is called a C compiler.

This C compiler accepts C source modules as inputs and generates object modules or assembler source modules as outputs. Therefore, the programmer 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 this C compiler is illustrated in Figure 1-1.

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.

The software required in connection with this C compiler is shown below.

- Editor ..... for source module file creation
- RA78K4 assembler package

Assembler ..... for converting assembly language into machine language Object converter ..... for conversion to HEX-format object module files Linker..... for linking object module files Librarian ..... for creating library files

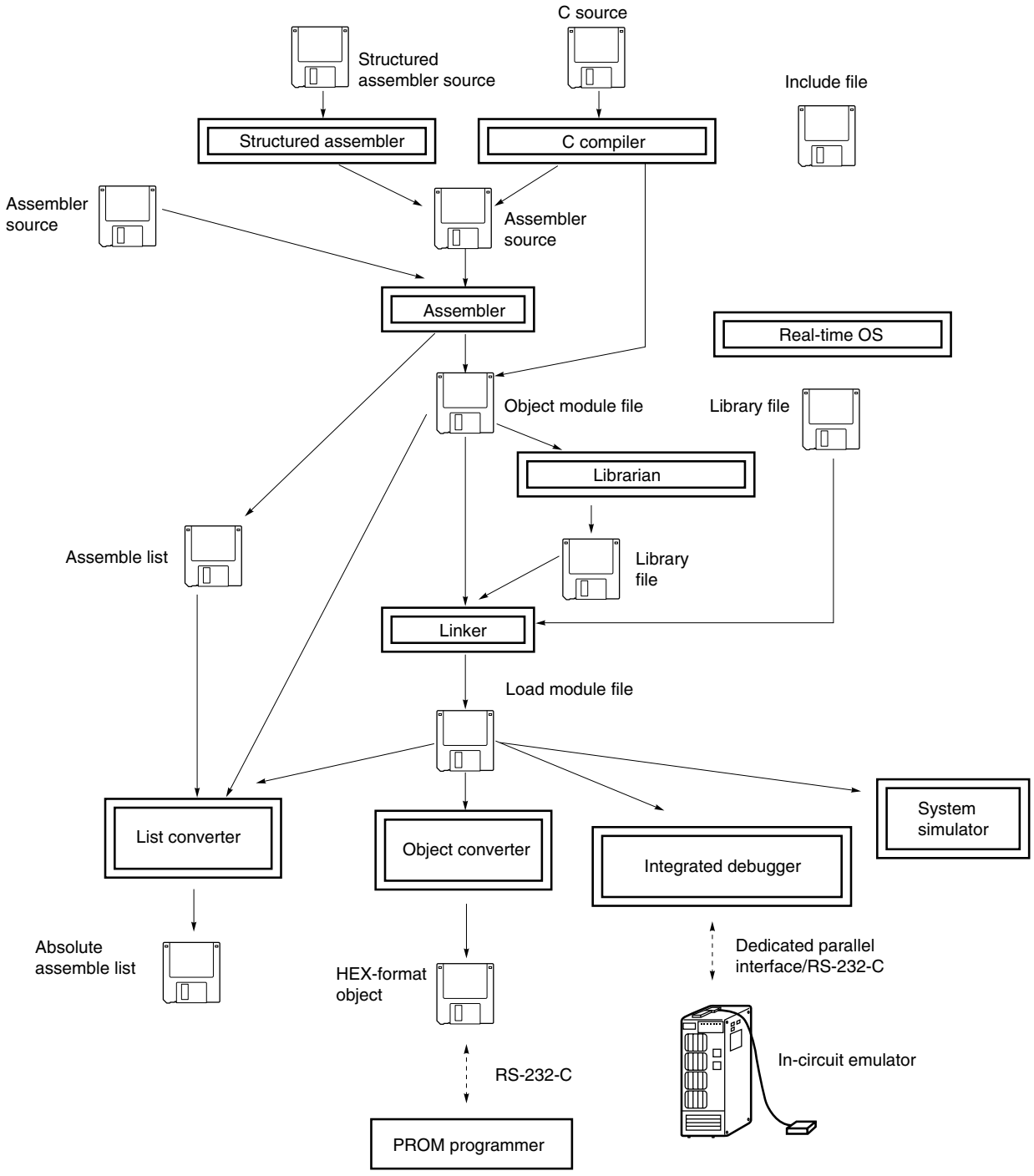
- Debugger (for 78K/IV) ..... for debugging C source module files

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

<1> Divides 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, this 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 This C Compiler

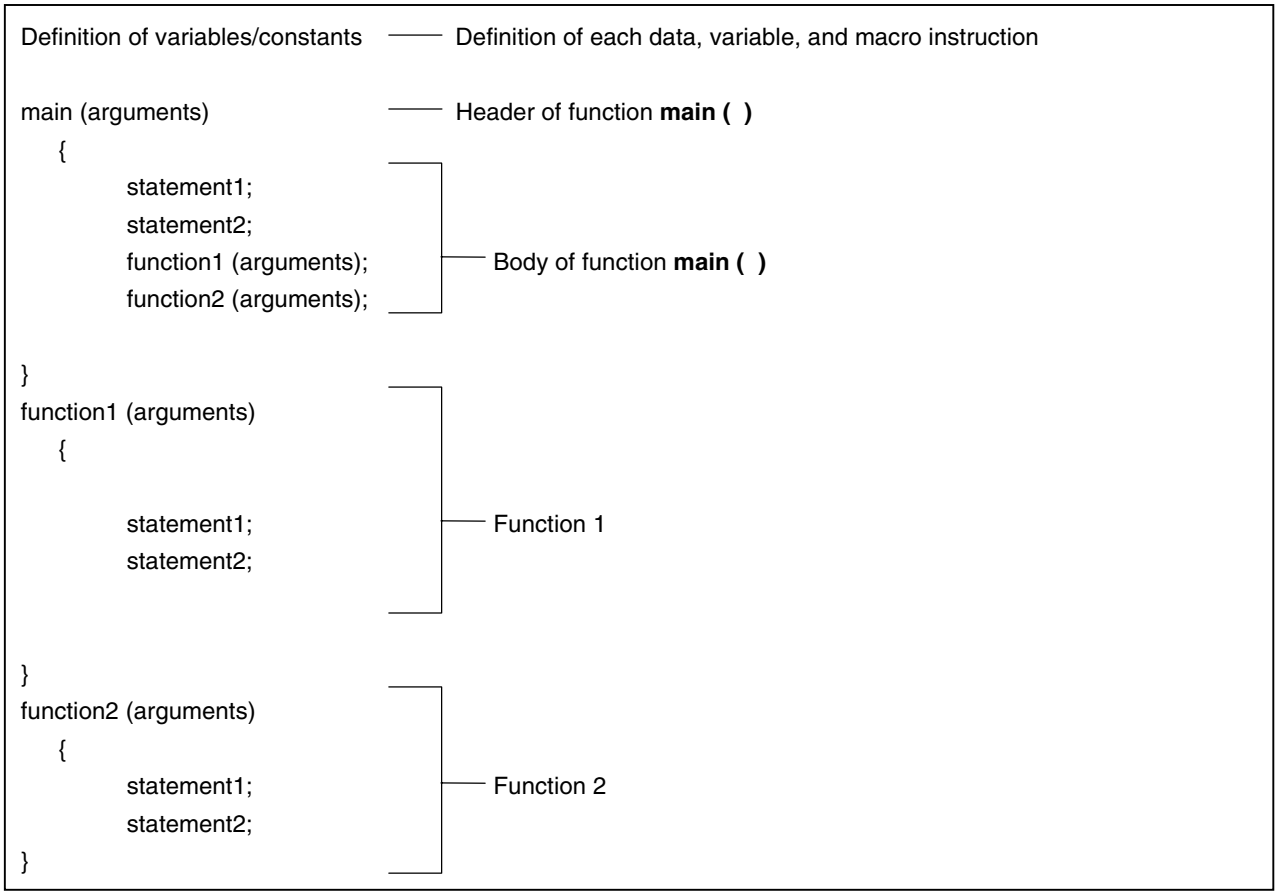


1.3 Basic Structure of C Source Program

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 FALSE 0                          ├─ #define  xxx xxx ..... <6> Preprocessing (macrodefinition)
#define SIZE  200                        ┘

void printf (char *, int);               ┐
void putchar (char);                     ┘  xxx xxxx (xxx, xxx) .....<7> Function prototype declarator

char  mark[SIZE+1];                      ┐  char xxx          .....<1> Type declarator, <5> External definition
                                         ┘
main ( )                                    xx [xx]          ..... <2> Operator
{
   int i,prime, k, count;                    int xxx  ..... <1> Type declarator
   count = 0;                                xx = xx  ..... <2> Operator
   for (i = 0; i <= SIZE;i++)                for (xx;xx;xx) xxx ; .....<3> Control structure
      mark[i] = TRUE;
   for (i = 0; i <= SIZE ; i++) {
      if (mark[i]) {
            prime = i + i + 3;             xxx = xxx + xxx + xxx.....<2> Operator
            printf ("%6d", prime);         xxx (xxx);.....<2> Operator
              count++;
            if ((count%8) = = 0) putchar ('\n');
 .....                                     if {xxx) xxx ; ..... <3> Control structure
             for (k = i + prime ; k <= SIZE ; k += prime)
                    mark [k] = FALSE;
       }
   }
   printf ("\n%d primes found. ", count);  xxx (xxx) ; ..... <2> 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 that instruct the compiler to perform an arithmetic operation, logical operation, assignment, etc. 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 the 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 one 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> Preprocessing

This is an instruction for the compiler. **#define** instructs the compiler to replace a parameter that is the same as the first operand with the second operand if the parameter appears in the program. For details, see **CHAPTER 9 PREPROCESSINGS (COMPILER DIRECTIVES)**.

<7> Declaration of function prototype

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

1.4 Reminders Before Program Development

Before commencing program development, keep in mind the points (limit values or minimum guaranteed values) summarized in Table 1-1 below.

Table 1-1. Maximum Performance Characteristics of This C Compiler

No.	Item	Limit Value/Min. Guaranteed Value
1	Nesting of compound statements, looping statements, or conditional control statements	45 levels
2	Nesting of conditional translations	255 levels
3	Number of arithmetic types, structure types, pointer to qualify union types or incomplete types, arrays, and function declarators in a declaration (or any combination of these)	12 levels
4	Nesting of parentheses per expression	32 levels
5	Number of characters that have a meaning as a macro name	256 characters
6	Number of characters that have a meaning as an internal or external symbol name	249 characters
7	Number of symbols per source module file	1,024 symbols[Note 1]
8	Number of symbols that have block scope within a block	255 symbols[Note 1]
9	Number of macros per source module file	10,000 macros[Note 2]
10	Number of parameters per function definition or function call	39 parameters
11	Number of parameters per macro definition or macro call	31 parameters
12	Number of characters per logical source line	2048 characters
13	Number of characters within a string literal after linkage	509 characters
14	Size of one data object	65,535 bytes
15	Nesting of **#include** directives	8 levels
16	Number of **case** labels per **switch** statement	257 labels
17	Number of source lines per translation unit	Approx. 30,000 lines
18	Number of source lines that can be translated without temporary file creation	Approx. 300 lines
19	Nesting of function calls	40 levels
20	Number of labels within a function	33 labels

No.	Item	Limit Value/Min. Guaranteed Value
21	Total size of code, data, and stack segments per object module	65,535 bytes[Note 3]
22	Number of members per structure or union	256 members
23	Number of **enum** constants per enumeration	255 constants
24	Nesting of structures, unions inside a structure or union	15 levels
25	Nesting of initializer elements	15 levels
26	Number of function definitions in 1 source module file	1,000
27	Level of the nest of declarator enclosed with parentheses inside a complete declarator.	591
28	Nesting of macros	200 levels
29	Number of –I include file path specifications	64

Notes 1. This value applies when symbols can be processed with the available memory space alone without using any temporary file. When a temporary file is used because of insufficient memory space, this value must be changed according to the file size.

2. This value includes the reserved macro definitions of the C compiler.

3. The large model provides 1,024 KB of code segments and 16 MB of data and stack segments altogether (when the **–ML** option is specified). The medium model provides 1,024 KB of code segments and 64 KB of data and stack segments altogether (when the **–MM** option is specified). The location (–CS0 or –CS15) can be specified for both models (the default is large model, location 0FH).

1.5 Features of This C Compiler

This C compiler has extended functions for CPU code generation that are not supported by ANSI (American National Standards Institute) Standard C. The extended functions of the C compiler allow the special function registers for the 78K/IV Series to be described at the C language level and thus help shorten object code and improve program execution speed. For details of these extended functions, see **CHAPTER 11 EXTENDED FUNCTIONS** in this manual.

Outlined here are the following extended functions that help shorten object code and improve execution speed.

- **callt** /_ **_callt** functions ..... Functions can be called using the **callt** table area.
- Register variables ..... Variables can be allocated to registers.
- **sreg**/_ **_sreg**/_ **_sreg1** variablesVariables can be allocated to the **saddr** area.
- **sfr** area ..... **sfr** names can be used.
- **noauto** functions..... Functions that do not output code for stack frame formation can be
  norec/_ **_leaf** functions..... created.
- ASM statements..... An assembly language program can be described in a C source program.
- **bit** type variables,..... Accessing the **saddr** or **sfr** area can be made on a bit-by-bit basis.
  boolean/_ **_boolean** type variables,
  _ **_boolean**1 type variables
- **callf**/_ **_callf** functions ..... A function body can be stored in the **callf** area.
- Bit field declaration ..... A bit field can be specified with **unsigned char** type.
- Multiplication function..... The code to multiply can be directly output with inline expansion.
- Division function..... The code to divide can be directly output with inline expansion.
- Rotate function..... The code to rotate can be directly output with inline expansion.
- Absolute address function..... Specific addresses in the memory space can be accessed.
- Data insertion function ..... Specific data and instructions can be directly embedded in the code area.
- _ _pascal function ..... The used stack is corrected on the called function side.
- Memory manipulation function ..... **memcopy** and **memset** can be directly output with inline expansion.
- **callf** two-step branch function ..... A two-step branch function is performed in the **callf** area.
- Three-byte address
  reference/generation function ..... Three-byte address reference/generation is performed.

An outline of the extended functions of this compiler is shown below. For details of each extended function, refer to **CHAPTER 11**.

<1> **callt**/_ **_callt** functions

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.

<2> Register variables

Variables declared with the **register** storage class specifier are allocated to the register or **saddr** area. Instructions to the variables allocated to a register or **saddr** area are shorter in code length than those to memory. This helps shorten object and improves program execution speed as well.

<3> Using the **saddr** area

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 (only to the **saddr2** area).

<4> **sfr** area

By declaring use of **sfr** names, manipulations on the **sfr** area can be described at the C source file.

<5> **noauto** functions

Functions declared as **noauto** do not output code for preprocessing and postprocessing (stack frame formation). By calling a **noauto** function, arguments are passed via registers. This helps shorten object code and improve program execution speed as well. This function has restrictions on arguments/automatic variables. For the details, refer to **11.5 (5) noauto function**.

<6> **norec**/_ _**leaf** functions

Functions declared as **norec**/_ _**leaf** do not output code for preprocessing and postprocessing (stack frame formation). By calling a **norec**/_ _**leaf** function, arguments are passed via registers as much as possible. Automatic variables to be used inside a **norec**/_ _**leaf** function are allocated to register or the **saddr** area. This helps shorten object code and also improve program execution speed. This function has restrictions on arguments/automatic variables and is not allowed to call a function. For the details, refer to **11.5 (6) norec function**.

<7> **bit** type variables and **boolean**/_ _**boolean** type variables

Variables with a 1-bit storage area are generated. By using the **bit** type variable or **boolean/_ _ boolean** type variable, the **saddr2** 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.

<8> **boolean1** type variables

Variables with a 1-bit storage area are generated. By using the _ _ **boolean1** type variable, the **saddr1** area can be accessed in bit units.

The _ _**boolean1** type variable is the same as the **bit** type variable in terms of both function and usage.

<9> ASM statements

The assembler source program described by the user can be embedded in an assembler source file to be output by this C compiler.

<10> Interrupt functions

A vector table and an object code corresponding to the interrupt are output. This allows programming of interrupt functions at the C source level.

<11> Interrupt function qualifier

This qualifier allows the setting of a vector table and interrupt function definitions to be described in a separate file.

<12> Interrupt function

An interrupt disable instruction and an interrupt enable instruction are embedded in an object.

<13> CPU control instructions

Each of the following instructions is embedded in an object.

- Instruction to set the value for halt to the STBC register
- Instruction to set the value for stop to the STBC register
- brk instruction
- nop instruction

<14> **callf**/_ _**callf** function

The **callf** instruction stores the body of a function in the callf entry area and allows the calling of the function with a code shorter than that with the **call** instruction. This improves executing speed and shortens the object code.

<15> Usage of 16 MB expansion space

Object files that linearly access the 16 MB expansion space are generated by an option.

<16> Location function

The location of the **saddr** area can be changed by an option if the memory model is large or medium.

<17> Absolute address access function

Codes that access the ordinary memory space are created with direct inline expansion without resort to a function call, and an object file is created.

<18> 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.

<19> Function to change compiler output section name

By changing the compiler section output name, the section can be independently allocated with a linker.

<20> Binary constant description function

Binary can be described in the C source.

<21> Module name change functions

Object module names can be freely changed in the C source.

<22> Rotate function

The code to rotate the value of an expression to the object can be directly output with inline expansion.

<23> Multiplication function

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.

<24> Division function

The code to divide 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.

<25> Data insertion function

Constant data is inserted in the current address. Specific data and instructions can be embedded in the code area without using assembler description.

<26> Interrupt handler for RTOS

Interrupt handlers for the RX78K/IV (real-time OS) can be described. Vectors can be set (settings of interrupt request name, function name for handlers, and stack switching) by the **#pragma** directive.

<27> Interrupt handler qualifier for RTOS

This qualifier allows the interrupt handler description and the vector setting for the RX78K/IV (real-time OS) to be made in separate files.

<28> Task function for RTOS

Specified functions are interpreted as the tasks for the RX78K/IV (real-time OS) by the **#pragma** directive. This allows the description of task function for RTOS with better code-efficiency at the C source level.

<29> Changing function call interface

Arguments can be passed by the previous function interface specification (using the stack only, with CC78K4 Ver.1.00 compatibles) by specifying the **-ZO** option during compilation.

<30> Change of calculation method of offset of arrays and pointers

The code efficiency is improved by performing an unsigned index calculation for the offset of the arrays and pointers (distance from the start of the array or pointer).

<31> Pascal function (_ _**pascal**)

The stack correction used to place arguments during the function call is performed on the called function side, not on the side calling the function. This shortens the object code when there are function calls in many places.

<32> Automatic pascal functionization of function call interface

_ _**pascal** attributes are added to all functions that can be pascal functionized.

<33> Flash area allocation method

Object files to be allocated to the flash area are generated.

<34> Flash area branch table

Startup routines and interrupt functions can be allocated to the flash area.

A function can be called from the boot area to the flash area.

<35> Function call function from boot area to flash area

A function in the flash area can be called from the boot area.

<36> Firmware ROM function

Manipulations regarding the firmware ROM function can be described at the C source level.

<37> Limiting **int** expansion of argument/return value

When the argument/return value of a function has the **char/unsigned** type, object files that do not perform **int** expansion are generated. This method can shorten the object code and improve the execution speed.

<38> Memory manipulation function

Memory manipulation functions can be output to an object directly with inline expansion. This function can shorten the object code and improve the execution speed.

<39> **callf** two-step branch function

Compared when a function body is allocated in the **callf** area, the **callf/_ _callf** attribute can be added to many more functions. Therefore, this function can shorten the object code if many functions that include call function are frequently used.

<40> Automatic **callf** functionization of function call interface

The _ _**callf** attribute is added to all functions except for the **callt/_ _callt/_ _interrupt/_ _interrupt_brk/_ _rtos_interrupt** functions.

<41> Three-byte address reference/generation function

Three-byte address reference/generation can be performed with a short code without using a complex cast description.

<42> Absolute address allocation specification

The external variable that declared _ _**directmap** and a static variable in a function can be allocated to any address, and multiple variables can be allocated in duplicate to the same address.

CHAPTER 2 CONSTRUCTS OF C LANGUAGE

This chapter explains the constituent 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 a C program description example are shown below.

```
#include "expand. h"
extern void testb (void);

extern void chgb (void);
extern bit data1;
extern bit data2;

void main ( )
{
       data1 = 1 ;
       data2 = 0 ;

       while(data1) {
              data1 = data2  ;
              testb( ) ;
       }
       if (data1 && data2) {
              chgb ( ) ;
       }
}

void lprintf (char *s, int I)
{
    int j;
    char *ss;
    j = i;
    ss = s;
}
    .
    .
    .
```

```
extern ..... Keyword

data1, data2 ..... Identifiers

void ..... Keyword

1 ..... Constant
0 ..... Constant

while ..... Keyword
{ } ..... Delimiter
= ..... Operator

if ..... Keyword
&& ..... Operator
( ) ..... Operator

lprintf ..... Identifier
char, int ..... Keywords
s, i ..... Identifiers
* ..... Operator
```

2.1 Character Sets

(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.

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 decimal numbers

```
0  1  2  3  4  5  6  7  8  9
```

29 graphic characters

```
!  "  #  %  &  '  (  )  *  +  ,  -  .  /  :
;  <  =  >  ?  [  ¥  ]  ^  —  {  |  }  ~
```

and nonprintable control characters which indicate space, horizontal tab, vertical tab, form feed, etc.

Remark In character constants, string literals, and comment statements, characters other than the above may also be used.

(2) Escape sequences

Nongraphic characters used for control characters such as alert, form feed are represented by escape sequences. Each escape sequence consists of a backslash (\) and a letter.

Nongraphic characters represented by escape sequences are shown below.

Table 2-1. List of Escape Sequences

Escape Sequence	Meaning	Character Code
\a	Alert	07H
\b	Backspace	08H
\f	Form feed	0CH
\n	Line feed	0AH
\r	Carriage return	0DH
\t	Horizontal tab	09H
\v	Vertical tab	0BH

(3) Trigraph sequences

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

Table 2-2. List of Trigraph Sequence

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

2.2 Keywords

(1) ANSI keywords

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

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) Keywords added for the CC78K4

In this C compiler the following tokens have been added as keywords to implement its expanded functions. As with ANSI keywords, hese tokens cannot be used as labels or variable names (when an uppercase character is included, the token is not regarded as a keyword).

Keywords that do not start with "_ _" can be made invalid by specifying the option that enables only ANSI-C language specification (**–ZA**).

_ _callt/callt	Declaration of **callt** function
_ _callf/callf	Declaration of **callf** function
_ _sreg/sreg	Declaration of **sreg** variable
_ _sreg1	Declaration of **sreg1** variable
noauto	Declaration of **noauto** function
_ _leaf/norec	Declaration of **norec** function
bit	Declaration of **bit** type variable
_ _boolean/boolean	Declaration of **boolean** type variable
_ _boolean1	Declaration of **boolean1** type variable
_ _interrupt	Hardware interrupt function
_ _interrupt_brk	Software interrupt function
_ _asm	asm statement
_ _rtos_interrupt	Interrupt handler for RTOS
_ _pascal	Pascal function
_ _flash	Firmware ROM function
_ _directmap	Absolute address allocation specification

2.3 Identifiers

An identifier is the name given to the following variables.

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 on the maximum length of the identifier. In this compiler, however, only the first 249 characters can be identified (refer to **Table 1-1 Maximum Performance Characteristics of This C Compiler**).

_(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 four types.

Function scope
File scope
Block scope
Function prototype scope

```
extern _ _ boolean data1, data2;  ———  data1, data2..... File scope
void testb(int x);                ———  x..... Function prtotype 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 beginning to 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 the case when the same identifier declared more than once in different scopes or in the same scope can be referenced as the same object or function. By being linked an identifier is regarded to be one and the same. An identifier may be linked in the following three 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 are examples of external linkage.

- The identifier of a function declared without a storage class specification.
- The identifier of an object 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 one translation (compiling) unit.

The following identifier is an example of internal linkage.

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

2.3.3 Name space for identifiers

All identifiers are classified into the following “name spaces”.

- 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 two 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 that have a static storage duration are as shown below.

- Objects that have external linkage
- Objects that have internal linkage
- Objects declared by the storage class specifier **static**

(2) Automatic storage duration

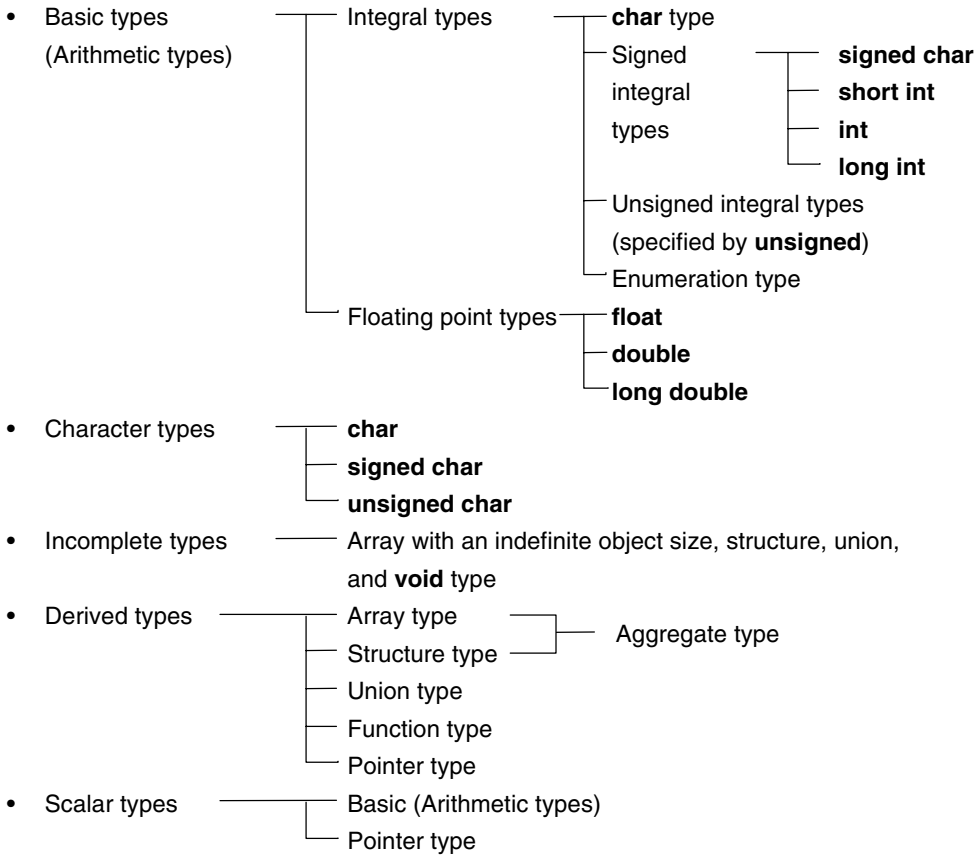
For objects that have automatic storage duration, an area is reserved when they enter a block to be declared.
If initialization is specified, the objects are initialized as they enter from the beginning of the block. In this case, if any object enters the block by jumping to a label within the block, 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 that have no linkage
- Objects declared inside a block without the storage class specifier **static**

2.3.5 Data types

A **type** determines the meaning of the value to be stored in each object.
Data types are divided into the following three categories depending on the variable to be declared.

- Object type ..... Type that indicates an object with size information
- Function type ..... Type that indicates a function
- Incomplete type ..... Type that indicates an object without size information



(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 types.

(a) Integral types

Integral data types are subdivided into four 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

(i) **char** type

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

(ii) Signed integral type

The signed integral type is subdivided into the following four 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 a qualifier.

An **int** object without a 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.

(iii) Unsigned integral data

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 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.

(iv) Enumeration type

Enumeration is a collection or list of named integer constants. An enumeration type consists of one or more sets of enumeration.

(b) Floating-point types

The floating-point types are subdivided into three types.

- **float**
- **double**
- **long double**

In this compiler, **double** and **long double** types as well as the **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.

Table 2-3. List of Basic Data Types

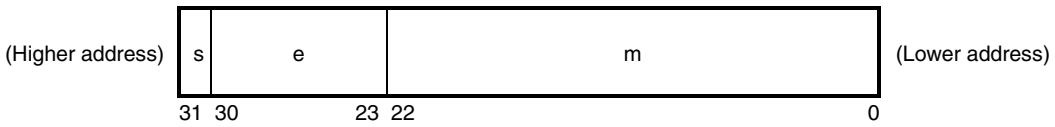
Type	Value Range
(**signed**) **char**	–128 to +127
unsigned char	0 to 255
(**signed**) **short int**	–32768 to +32767
unsigned short int	0 to 65535
(**signed**) **int**	–32768 to +32767
unsigned int	0 to 65535
(**signed**) **long int**	–2147483648 to +2147483647
unsigned long int	0 to 4294967295
float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

- The **signed** keyword may be omitted. However, with the **char** type, it is judged as **signed char** or **unsigned char** depending on the condition at compilation.
- **short int** data and **int** data are handled as data that have the same value range but are of different types.
- **unsigned short int** data and **unsigned int** data are handled as data that have the same value range but are of different types.
- **float**, **double**, and **long double** data are handled as data that have the same value range but are of different types.

(i) Floating-point number (float type) specifications

- 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: Exponent (8 bits)

An exponent with a base of 2 is expressed as a 1-byte integer (expressed by two's complement in the case of a negative), and used after having a further bias of 7FH added. These relationships are shown in Table 2-4 below.

Table 2-4. Exponent Relationships

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

m: Mantissa (23 bits)

The mantissa is expressed as an absolute value, with bit positions 22 to 0 equivalent to the 1st to 23rd places of a binary number. Except for when the value of the floating point is 0, the value of the exponent is always adjusted so that the mantissa is within the range of 1 to 2 (normalization). The result is that the position of 1 (i.e. the value of 1) is always 1, and is thus represented by omission in this format.

- Zero expression

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

(Value of sign)	
(–1)	* 0

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

$$(-1)^{(\text{Value of sign})} * \infty$$

- Unnormalized value
  When exponent = 0 and mantissa ≠ 0, the unnormalized value is expressed as follows.

$$(-1)^{(\text{Value of sign})} * (\text{Value of mantissa}) * 2^{-126}$$

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.

- Not-a-number (NaN) expression
  When exponent = FFH and mantissa ≠ 0, NaN is expressed, regardless of the sign.

- 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 two values that can express the differential of the prerounded value, round to an even number (a number whose lowest binary bit is 0).

- Operation exceptions
  There are five types of operation exceptions, as shown below.

Table 2-5. List of Operation Exceptions

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) Character types

The character data types include the following three types.

- **char**
- **signed char**
- **unsigned char**

(3) Incomplete types

The incomplete data types include the following four types.

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

(4) Derived types

The derived types are divided into the following three categories.

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

(a) Aggregate type

The aggregate type is subdivided into two types.

Array type and Structure type. An aggregate type data is a collection of member objects to be taken successively.

i) Array type

The array type continuously allocates a collection of member objects called element types. 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 an incomplete type array.

ii) Structure type

The structure type continuously allocates member objects each differing in size. Each member object can be specified by name.

(b) 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.

(c) Function type

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

(d) 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.

(5) Scalar types

The basic types (arithmetic types) 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

2.3.6 Compatible type and composite type

(1) Compatible type

If two types are the same, they are said to be compatible or have compatibility. For example, if two 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 two structures or unions must be in the same order and the individual members (enumerated constants) of the two enumerated types must have the same values.

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

(2) Composite type

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

- If either of the two 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 that 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 two prototypes.

[Example of composite type]

Assume that two 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] ) ;
```

2.4 Constants

A constant is a variable that does not change in value during the execution of the program, and its value must be set beforehand. The type for each constant is determined according to the format and value specified for the constant. The following four constant types are available.

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

2.4.1 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**)
	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).

2.4.2 Integer constant

An integer constant starts with a number and does not have a decimal point or 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.

There are the following three types of integer constants.

- Decimal constant: Decimal number that starts with a number other than 0
  Decimal number = `123456789`
- Octal constant: Integer suffix 0 + octal number
  Octal number = `01234567`
- Hexadecimal constant: Integer suffix 0x or 0X + hexadecimal number
  Hexadecimal number = `0123456789`
  `abcdef ABCDEF`

Unsigned suffix

`u U`

Long suffix

`l L`

(1) Decimal constant

A decimal constant is an integer value with a base (radix) of 10 and must begin with a number other than 0 followed by any numbers 0 through 9 (example: 56UL).

(2) Octal constant

An octal constant is an integer value with a base of 8 and must begin with 0 followed by any numbers 0 through 7 (example: 034U).

(3) Hexadecimal constant

A hexadecimal constant is an integer value with the base of 16 and 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).

The type of integer constant is regarded as the first of the “representable type” shown below.

In this compiler, the type of the unsubscripted constant can be changed to **char** or **unsigned char** depending on the compile condition (option).

(Integer constant) (Representable type)

- Unsuffixed decimal number ..... **int**, **long int**, **unsigned long int**
- Unsuffixed octal, hexadecimal number..... **int**, **unsigned int**, **long int**, **unsigned long int**
- Suffixed u or U..... **unsigned 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**

2.4.3 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 three 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.

2.4.4 Character constants

A character constant is a character string that includes one or more characters enclosed in a pair of single quotes as in 'X' or 'ab'.

A character constant does not include single quote', backslash (¥ or \), and line feed character (¥n). To represent these characters, escape sequences are used. There are the following three 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])

Notes 1. Null character

2. In this compiler, \xFF represents –1. If the condition (option) that regards **char** as **unsigned char** is added, however, it represents +255.

2.5 String Literals

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 the escape sequence \’, whereas a double quote (”) is represented by the escape sequence \”.

Array elements have a **char** type string literal and are initialized by assigned tokens (example: char array [ ] = “abc”;).

2.6 Operators

The operators are shown below.

```
[ ]   ( )   .     ->
++    - -   &     *     +     –     ~     !     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 preprocessings. (For the description, refer to **CHAPTER 5 OPERATORS AND EXPRESSIONS**.)

2.7 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.

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

An expression declaration or statement may be described in brackets “[ ]”, parentheses “( )”, or braces “{ }”, These delimiters must always be used in pairs as shown above. The delimiter # is used only for preprocessings.

2.8 Header Name

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

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

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

2.9 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 part after "//" to the line feed can be identified as a comment statement using the **–ZP** option.

Example /* comment statement */
//comment statement

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 the 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.

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

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

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

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

If an identifier for an object is declared to have no linkage, the type of the object must be perfect (the object with information related to the size) at the end of the declarator or initialize declarator (if it has an 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 that the object has, and the scope of the object. In a declaration, only one storage class specifier can be described. The following five storage class specifiers are available.

- `typedef`
- `extern`
- `static`
- `auto`
- `register`

(1) typedef

The **typedef** specifier declares a synonym for the specified type. See 3.6 below for details of the **typedef** specifier.

(2) extern

The **extern** specifier indicates (tells the compiler) that the variable immediately before this specifier is declared elsewhere in the program (i.e., an external variable).

(3) static

The **static** specifier indicates that an object has static storage duration. For an object that has static storage duration, an area is reserved before the program execution and the value to be stored is initialized only once. The object exists throughout the execution of the entire program and retains the value last stored in it.

(4) 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 that has automatic storage duration will not be guaranteed after the execution of the declared block.

(5) register

The **register** specifier indicates that an object is assigned to a register of the CPU. With this C compiler, 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` or `union` specifier
- `enumeration` specifier
- `typedef` name

In this C compiler, the following type specifiers have been added.

- `bit/boolean/_ _boolean/_ _boolean1`

The following explains the meaning of each type specifier and the limit values that can be expressed with this compiler (the values enclosed in the parentheses). Since this compiler 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.

- `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 (–32768 to +32767)
- `unsigned short, unsigned short int` Unsigned integer (0 to 65535)
- `int, signed, signed int` ..... Signed integer (–32768 to +32767)
- `unsigned, unsigned int` ..... Unsigned integer (0 to 65535)
- `long, signed long, long int, signed long int` ..... Signed integer (–2147483648 to +2147483647)
- `unsigned long, unsigned long int` ... Unsigned integer (0 to 4294967295)
- `float` ..... Single precision floating-point number (1.17549435E–38F to 3.40282347E+38F)
- `double` ..... Double precision floating-point number (1.17549435E–38F to 3.40282347E+38F)
- `long double` ..... Extended precision floating-point number (1.17549435E–38F to 3.40282347E+38F)
- `structure/union` specifier ..... Collection of member objects
- `enumeration` specifier ..... Collection of int **type** constants
- `typedef` name ..... Synonym of specified type
- `bit/boolean/_ _boolean/_ _boolean1` Integers represented with a single bit (0 to 1)

Type specifiers delimited with a comma have the same size.

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 one 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 declaration.
Align data is inserted by specifying the **-RP** option.

A 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 one object. Each type of object is called a member and can be given a name. The members of a union overlap each other in area, namely, they share the same area.

A union is declared 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. A member can be declared with the number of bits specified. A member with the number of bits specified is called a bit field.
In this compiler, extended functions related to bit field declaration have been added. For details, refer to **11.5 (19) 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] The MSB of an **int** field which has no qualifier or a **signed int** field will be judged as a sign bit.[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.

The unary & operator (address) cannot be applied to a bit field object.

Notes 1. In this compiler, **char** type, **unsigned char** type, and **signed char** type can also be specified. All of them are regarded as **unsigned** type since this compiler does not support **signed** type bit fields.

2. In this compiler, the direction of bit field allocation can be changed using the compiler option **–RB** (for details, refer to **CHAPTER 11 EXTENDED FUNCTIONS)**.

The following shows an example of a bit field.

```
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 is declared as shown below.

```
enum [identifier] {enumerator list}
```

Objects are declared according to 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 by “=”.

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 ;
        /*...*/ (*cp != burgundy) /*...*/
              :
```

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.

The identifier in the following declaration is a tag name.

```
structure/union   identifier {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.

```
struct/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 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 two 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

Two type qualifiers are available: **const** and **volatile**. These type qualifiers affect left-side values only.

Using a left-side value that has a non-**const** type qualifier cannot change an object that has been defined with a **const** type qualifier. Using a left-side value that has a non-**volatile** type qualifier cannot reference an object that has been defined with a **volatile** type qualifier.

An object that has a **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 stored last in the object at every sequence point must be in agreement with those determined by the program, except for the changes due to 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,** **__pascal**, which are the type qualifiers unique to this compiler mentioned in **2.1 Keywords**, can be included as type qualifiers.

sreg, **__sreg**, **__sreg1**, and **__directmap** are also type qualifiers.

In the following example, “real_time_clock” can be changed by hardware, but operations such as assignment, increment, and decrement are not possible.

```
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 left-side value 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. The scope of an identifier and a function or object that has a storage duration and a type are determined by a declarator.

A description of each declarator is given 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 declaration is performed as follows.

```
* type qualifier list   identifier
```

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

The following two 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 two 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 that 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 list. By these lists, an identifier, which indicates the 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.

- int ..... Specifies an **int** type.
- int * ..... Specifies a pointer to an **int** type.
- int *[3] ..... Specifies an array that has three pointers to an **int** type.
- int (*) [3] ..... Specifies a pointer to an array that has three **int** types.
- int *( ) ..... Specifies a function that returns a pointer to an **int** type that has no parameter specification.
- int (*) (void) ..... Specifies a pointer to a function that returns an **int** type that no parameter specification.
- int (*const [ ]) (unsigned int, ...) .... Specifies an indefinite number of arrays that have one 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, the **typedef** name t is declared with a **signed int** type, and the **typedef** name plain is declared with an **int** type, and a structure with three 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 two bit field declarations differ in 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 (qualifies t which can be referred to as the **typedef** name). After this declaration, if:

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

is found within the effective range, the function f is declared as “function that has one parameter and returns **signed int**”, and the parameter is declared as “pointer type for the function that has one parameter and returns **signed int**”. The identifier t is declared as **long** type.

typedef names may be used to facilitate program reading. For example, the following three 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.

(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 that 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 */
}
```

(2) Initialization of objects that have an automatic storage duration

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

Example

```
void func(void){
        char   aval;            /*undefined at this point */
           :
        aval = 1;               /* initialized to 1 */
}
```

(3) Initialization of character arrays

A character array can be initialized by a **char** string literal (**char** string enclosed with " "). Likewise, a character string in which a series of **char** string literals 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 a **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 a character string literal so that the length indicates 4 "**char** array" type objects.

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

(4) Initialization of aggregate or union type objects

- Aggregate type
  An aggregate type object is initialized by 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 that has static storage duration.
  With an array of 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.
- Union type
  A union type object is initialized by 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 one-dimensional array that has three 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 three-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 three-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 two 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. **Table 4-1** gives a list of conversions between types.

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	+	＼	○	○	○	○	○	○	○	○	○	○
	–	＼	N	○	N	○	N	○	N	○	○	○
unsigned char		Δ	＼	○	○	○	○	○	○	○	○	○
(signed) short int	+			＼	○	＼	○	○	○	○	○	○
	–			＼	N	＼	N	○	N	○	○	○
unsigned short int				Δ	＼	Δ	＼	○	○	○	○	○
(signed) int	+			＼	○	＼	○	○	○	○	○	○
	–			＼	N	＼	N	○	N	○	○	○
unsigned int				Δ	＼	Δ	＼	○	○	○	○	○
(signed) long int	+							＼	○	○	○	○
	–							＼	N	○	○	○
unsigned long int								Δ	＼	○	○	○
float										＼	○	○
double											＼	＼
long double											＼	＼

Remarks 1 The **signed** keyword may be omitted. However, with a **char** type data, the data type is regarded as the **signed char** or **unsigned char** type depending on the condition (option) for compilation.

2 Legend:

○: Type conversion will be performed properly.

\: Type conversion will not be performed.

N: 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.

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.

In general integral promotion, the value of the original data type is retained, including its sign. **char** type data without a type qualifier will normally be handled as **signed char** in this compiler.

If can also be handled as **unsigned char** by using an 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 by 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 to 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 the non-negative remainder of the value 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 overflowed value is ignored if the value after conversion cannot be expressed. For the conversion pattern, refer to **Table 4-1**. List of Conversions between Types.

Conversion operations from signed integral type to unsigned integral type are as listed in Table 4-2 below.

Table 4-2. Conversions from Signed Integral Type to Unsigned Integral Type

		unsigned	
		Smaller in Value Range	Greater in Value Range
signed	+	/	○
	–	/	+

○: 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

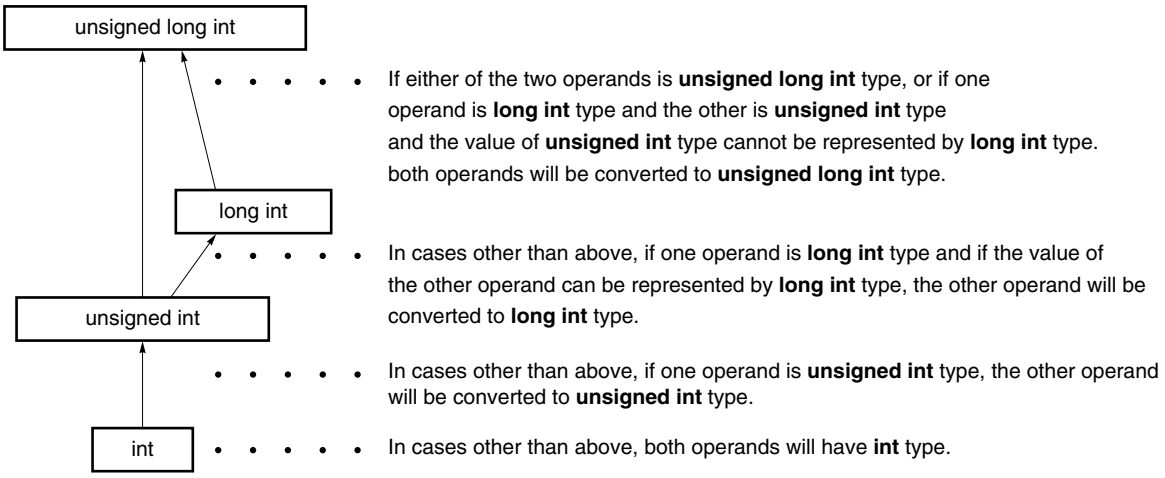
Types obtained as a result of operations on arithmetic type data have a wide range of values.

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 this compiler, the conversion to **int** type can be intentionally disabled by a compile condition (optimizing option) (For details, refer to **CC78K4 C Compiler Operation User's Manual (U15557E) CHAPTER 5 COMPILER OPTIONS**).

4.2 Other Operands

(1) Left-side values and function locators

A left-side value refers to an expression that specifies an object (and has an incomplete type other than object type or **void** type).

Left-side values that do not have array types, incomplete types, or **const** qualifier types, and structures or unions that have no **const** qualifier type members are “modifiable left-side values”.

A left-side value that 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 a left-side value.

The behavior of left-side values that have incomplete types but have no array types is not guaranteed.

A left-side value that has an “... array” type except character arrays will be converted to an expression that has a “pointer to ...” type. This expression is no longer a left-side value.

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 a context that 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 that 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');
          for  (k = i + prime ; k<=SIZE; k += prime)——— += ..... Assignment operator
                  mark [k] = FALSE;
          }
       }
```

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

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

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

Table 5-1 shows the evaluation priority of operators used in C.

Table 5-1. Evaluation Precedence of Operators

Type of Expression	Operator	Linkage	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

The arrow (→ or ←) in the “Linkage” column denotes that when an expression contains two or more operators of the same priority, 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 object or function
  (identifier primary expression)
- Constant (constant primary expression)
- String literal (constant primary expression)
- Expression enclosed in parentheses
  (parenthesized expression)

An identifier that becomes a primary expression is a left-side value 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 **2.4 Constants**. String literal(s) become a left-side value that has a data type as explained in **2.5 String Literals**.

5.2 Postfix Operators

A postfix operator is an operator that appears or is placed after an object or function.
The primary expressions are described below.

(1) Subscript operators

Postfix Operators — [ ] Subscript Operator

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, as many subscript operators 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 three members each consisting of five **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 $n \geq 2$) consisting of i*j*...*k, the array can be specified with n number of subscript operators. In this case, E becomes a pointer to an array of (n – 1)th dimension consisting of j*...*k.

SYNTAX

postfix-expression [subscripted expression]

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.

(2) Function call operators

Postfix Operators ( ) Function Call

FUNCTION

The postfix ( ) operator calls a function. The function to be called is specified with a postfix expression and argument(s) to be 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 a general integer. This is called “default actual argument extension”. Performing a function prototype declaration avoids default actual argument extension and detects errors in of the type and number of arguments and the type of return value.

Calling a function that has neither a storage class specification nor a data type specification such as “identifier ( );” is interpreted as calling a function that has an external object and returns an **int** type that has no information on arguments. In other words, the following declaration will be made implicitly.

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

SYNTAX

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

[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 a 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.

(3) Structure and union member

Postfix Operators . ->

<1> . (dot) operator

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.

SYNTAX

```
postfix-expression . identifier
```

<2> → (arrow) operator

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.

SYNTAX

```
postfix-expression → identifier
```

Postfix Operators . ->

[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) ;
}
```

(4) Postfix Increment/Decrement operators

Postfix Operators ++ – –

<1> Postfix ++ (Increment) operator

FUNCTION

The postfix ++ (Increment) operator increments the value of an object by 1. This increment operation is performed taking the data type of the object into account.

SYNTAX

postfix-expression ++

NOTE

See <2> below.

<2> Postfix – – (Decrement) operator

FUNCTION

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

SYNTAX

postfix expression – –

NOTE

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

5.3 Unary Operators

A unary operator performs an operation on one object or parameter (i.e., operand). The following unary operators are available.

- Prefix Increment and Decrement operators

  ```
  + + - -
  ```

- Address and Indirect operators

  ```
  & *
  ```

- Unary Arithmetic operators

  ```
  + - ~ !
  ```

- **sizeof** operator

  ```
  sizeof
  ```

The followings explain each unary operators are described below.

(1) Prefix Increment and Decrement operators

Unary Operators ++ – –

<1> Prefix ++ (Increment) operator

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
```

SYNTAX

```
+ + unary-expression
```

<2> Prefix – – (Decrement) operator

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
```

SYNTAX

```
- - unary-expression
```

(2) Address and Indirection operators

Unary Operators & *

<1> Unary & operator

FUNCTION

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

SYNTAX

```
& operand
```

<2> Unary * operator

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).

SYNTAX

```
* operand
```

NOTE

The operand of the unary & operator must be a left-side value 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.

(3) Unary Arithmetic operators (+ – ~ !)

Unary Operators + – ~ !

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.

SYNTAX

+	operand
–	operand
~	operand
!	operand

(4) sizeof operators

Unary Operators — sizeof Operator

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.

SYNTAX

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

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 a left-side value that refers to a bit field object cannot be used as the operand of this operator.

5.4 Cast Operators

A cast is a special operator that forces one data type to be converted into another. The cast operator is mainly used when converting a pointer type.

Cast Operators (type-name)

FUNCTION

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

SYNTAX

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

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

Arithmetic operators are divided into multiplicative operators and additive operators. Multiplicative operators find the product, quotient, and remainder of two operands. Additive operators find the sum and difference of two operands.

- Multiplicative operators `*` `/` `%`
- Additive operators `+` `-`

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

a/b		b	
		+	–
a	+	+	–
	–	–	+

a % b		b	
		+	–
a	+	+	+
	–	–	–

Remark a and b indicate operands.

Division is performed with two 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 two 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 two operands in division and remainder division, respectively. Multiplicative operators and additive operators are described below. E1 and E2 used in the explanation of syntax indicate operands or expressions.

(1) Multiplicative operators

Multiplicative Operators * / %

<1> * operator

FUNCTION

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

SYNTAX

```
E1 * E2
```

<2> / operator

FUNCTION

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

SYNTAX

```
E1 / E2
```

<3> % operator

FUNCTION

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

SYNTAX

```
E1 % E2
```

(2) Additive operators

Additive Operators + –

<1> + operator

FUNCTION

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

SYNTAX

```
E1 + E2
```

<2> – operator

FUNCTION

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

SYNTAX

```
E1 - E2
```

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. There are the following two shift operators.

- shift 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 and b indicate operands.

The shift operators are described below. E1 and E2 indicate operands or expressions.

Shift Operators << >>

<1> Left shift (<<) operators

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.

SYNTAX

```
E1 << E2
```

<2> Right shift (>>) operators

FUNCTION

The binary >> (right shift) operator shifts the left operand to the right the number of bits specified by the right operand. If the left operand is unsigned, zeros are filled in vacated bits (Logical shift). If the left operand is signed, a copy of the sign bit is filled in vacated bits.

If the left operand E1 is unsigned or signed and has a non-negative value in “E1>>E2”, the result will become a value obtained by dividing E1 by the E2th power of 2.

SYNTAX

```
E1 >> E2
```

5.7 Relational Operators

There are two types of operators to indicate the relationship between two operands: “relational operator” and “equality operator”.

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

The relational operators and equality operators are shown below.

- Relational operator `<` `>` `<=` `>=`
- Equality operator `==` `!=`

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

In this compiler, relational operators and equality operators generate ‘1’ if the specified relationship is true and ‘0’ if it is false. The results have int type.

The relational operators and equality operators are described below. E1 and E2 used in the explanation of syntax indicate operands or expressions.

(1) Relational operators

Relational Operators < > <= >=

<1> < (less than) operator

FUNCTION

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

SYNTAX

```
E1 < E2
```

<2> > (greater than) operator

FUNCTION

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

SYNTAX

```
E1 > E2
```

<3> <= (less than or equal) operator

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.

SYNTAX

```
E1 <= E2
```

<4> >= (greater than or equal) operator

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.

SYNTAX

```
E1 >= E2
```

(2) Equality operators

Equality Operators = = !=

<1> = = (equal) operator

FUNCTION

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

SYNTAX

```
E1 == E2
```

<2> != (not equal) operator

FUNCTION

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

SYNTAX

```
E1 != E2
```

5.8 Bitwise Logical Operators

Bitwise logical operators perform a specified logical operation on the value of an object in bit units. The bitwise logical expressions include Bitwise AND (&), Bitwise Exclusive OR (^), and Bitwise Inclusive OR (|).

Each logical operation is indicated by the operators shown below.

- Bitwise AND operator &
- Bitwise XOR operator ^
- Bitwise OR operator |

The bitwise logical operators are described below. E1 and E2 used in the explanation of syntax indicate operands or expressions.

(1) Bitwise AND operators

Bitwise AND Operators &

FUNCTION

The binary & operator is a bitwise **AND** operator that 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”.

Table 5-4. Bitwise AND Operation

		Value of Each Bit in Left Operand	
		1	0
Value of each bit in right operand	1	1	0
	0	0	0

SYNTAX

```
E1 & E2
```

(2) Bitwise XOR operators

Bitwise XOR Operators ^

FUNCTION

The binary ^ (caret) operator is a bitwise exclusive **OR** operator that 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.

Table 5-5. Bitwise XOR Operation

		Value of Each Bit in Left Operand	
		1	0
Value of each bit in right operand	1	0	1
	0	1	0

SYNTAX

```
E1 ^ E2
```

(3) Bitwise Inclusive OR operators

Bitwise Inclusive OR Operators |

FUNCTION

The binary | operator is a bitwise inclusive **OR** operator that 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.

Table 5-6. Bitwise OR Operation

		Value of Each Bit in Left Operand	
		1	0
Value of each bit in right operand	1	1	1
	0	1	0

SYNTAX

```
E1 | E2
```

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'. The following explains each logical operator. E1 and E2 used in the explanation of syntax indicate operands or expressions.

(1) Logical AND operators

Logical AND Operators &&

FUNCTION

The && operator performs a logical **AND** operation on two operands and returns a “1” if both operands have nonzero values. Otherwise, a “0” is returned. The type of the result is **int**.

Table 5-7. Logical AND Operation

		Value of Left Operand	
		Zero	Nonzero
Value of right operand	Zero	0	0
	Nonzero	0	1

SYNTAX

```
E1 && E2
```

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.

(2) Logical OR operators

Logical OR Operators ||

FUNCTION

The | | operator performs a logical **OR** operation on two operands and returns a “0” if both operands are zero. Otherwise, a “1” is returned. The type of result is **int**.

Table 5-8. Logical OR Operation

		Value of Left Operand	
		Zero	Nonzero
Value of each bit in right operand	Zero	0	1
	Nonzero	1	1

SYNTAX

```
E1 || E2
```

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 Operators

Conditional operators judge the processing to be performed next by the value of the first operand. Conditional operators judge by '?' and ':'. The conditional operators are described below.

(1) Conditional operators (?, :)

Conditional Operators ? :

FUNCTION

The conditional (?, :) operator evaluates the first operand before the ?. 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.

SYNTAX

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

EXAMPLE

```
#define TRUE  1
#define FALSE  0
char  flag ;
int  ret ;
ret  func () {
        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 a **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.

- Assignment Operators

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

The each assignment operators are described below. E1 and E2 used in the explanation of syntax indicate operands or expressions.

(1) Simple assignment operators

Simple Assignment Operators =

FUNCTION

The = (simple assignment) operator converts the right operand (expression) to the type of the left operand (left-side value) 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. 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 a 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 ;
/*...*/ ((c = f ()) == -1) /*...*/
```

SYNTAX

```
E1 = E2
```

(2) Compound assignment operators

Compound Assignment Operators

*= /= %= += −=
<<= >>= &= ^= |=

<1> Compound assignment operators

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 the left-side value (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 left-side value (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;
```

SYNTAX

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

5.12 Comma Operator

(1) Comma operator

Comma Operator ,

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);
}
```

SYNTAX

```
E1 , E2
```

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 instead of execution.

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.

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

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

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 complicated a process is, it can be expressed with three basic control structures. These three control structures are: Sequential, Conditional (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 (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 two 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 in the state indicated by the control statement.

(4) Branch processing

C is forced 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 six types of statements used in C.

• Labeled statement	Causes branch according to the value of the **switch** statement and the destination of the **goto** statement
• Compound statement (block)	Collects two or more statements to be processed as one unit
• Expression statement	A statement consisting of an expression and a semicolon
• Selection statement	Selects a statement out of several statements according to the value of the expression
• Iteration statement	Repeatedly performs a statement called the body of a loop until control expression becomes equal to 0
• Branch statement	Causes an unconditional branch to a different destination

A description example of these statements is shown below.

[Description example]

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

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

char   mark [SIZE+1];
void main (void) {
       int i, prime, k, count;

     count = 0 ;
        for (i = 0  ;  i <= SIZE  ;  i++) ..... /* for ..... Iteration statement */
               mark [i] = TRUE ;
        for (i = 0  ;  i <= SIZE  ;  i++)  {..... / * for ..... Iteration statement */
                 if  (mark [i])  { ..... / * if ..... Selection statement */
                       prime = i + i + 3;
                       lprintf ("%d"  , prime);
                       if ((count%8)  == 0)  putchar ('\n');
                       for (k = i + prime  ;
                            k <= SIZE  ;  k += prime)  / * if ..... Selection statement * /
                            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 the **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 syntax of labeled statements is given below.

(1) case label

Labeled Statements — case label

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.

SYNTAX

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

EXAMPLE 1

```
int  f (void), i;
void main (void) {
    /* ... */
    switch (f()) {
       case 1:
         i = i + 4 ;
          break ;
       case 2:
         i = i + 3 ;
        break ;
       case 3:
         i = i + 2 ;
    }
    /* ... */
}
```

EXPLANATION

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.

Labeled Statements — case label

EXAMPLE 2

```
int  i ;
void main (void) {
      /* ... */
    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.

(2) default label

Labeled Statements — default label

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.

SYNTAX

```
default : statement
```

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;
}
```

EXPLANATION

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 used 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 consists of two or more statements grouped together with enclosing braces and executed as one 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 does not need a 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 main(void) {
    /*...*/
    (void)p(0) ;
}
```

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

```
char *s ;
void main(void) {
    /*...*/
    while (*s++ != '0') ;
    /*   */
}
```

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

```
void func(void){
          /*...*/
     while (loop1) {
          /*...*/
          while (loop2)   {
               /*...*/
               if (want_out)
               goto end_loop1  ;
               /*...*/
          }
     end_loop1: ;
     }
}
```

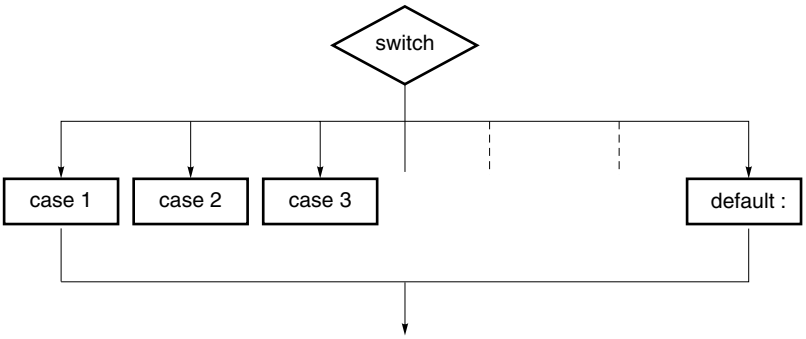
6.4 Conditional Statements

Conditional (or selection) statements include the **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.

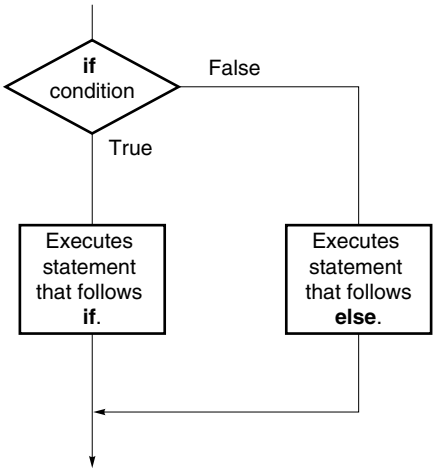
The control flows of **if** and **switch** statements are illustrated in Figure 6-1 below.

Figure 6-1. Control Flows of Conditional Statements

Control flow of **switch** statement



Control flow of **if** statement



(1) if and if ... else statements

Conditional Statements — if, if ... else

FUNCTION

An **if** statement has a one-way selection structure and executes the statement that follows the control expression enclosed in parentheses if the value of the control expression is nonzero (True).

An **if ... else** statement has a two-way selection structure and executes the statement-1 that follows the control expression if the value of the control expression is nonzero (True) or the statement-2 that follows **else** if the value of the control expression is zero (False).

SYNTAX

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

EXAMPLE

```
unsigned char    uc ;
void func (void){
    if ( uc < 10 ){
          /*111*/
     }
    else{
        /* 222 */
    }
}
```

EXPLANATION

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 the **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.

(2) switch statement

Conditional Statements switch

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.

SYNTAX

```
switch (expression) statement
```

EXAMPLE

```
extern void func(void);
unsigned char  mode ;
void main(void) {
    switch (mode) {
        case 2:
             mode = 8 ;
             break ;
        case 4:
             mode = 2 ;
             break ;
        case 8:
             func( );
    }
}
```

NOTE

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

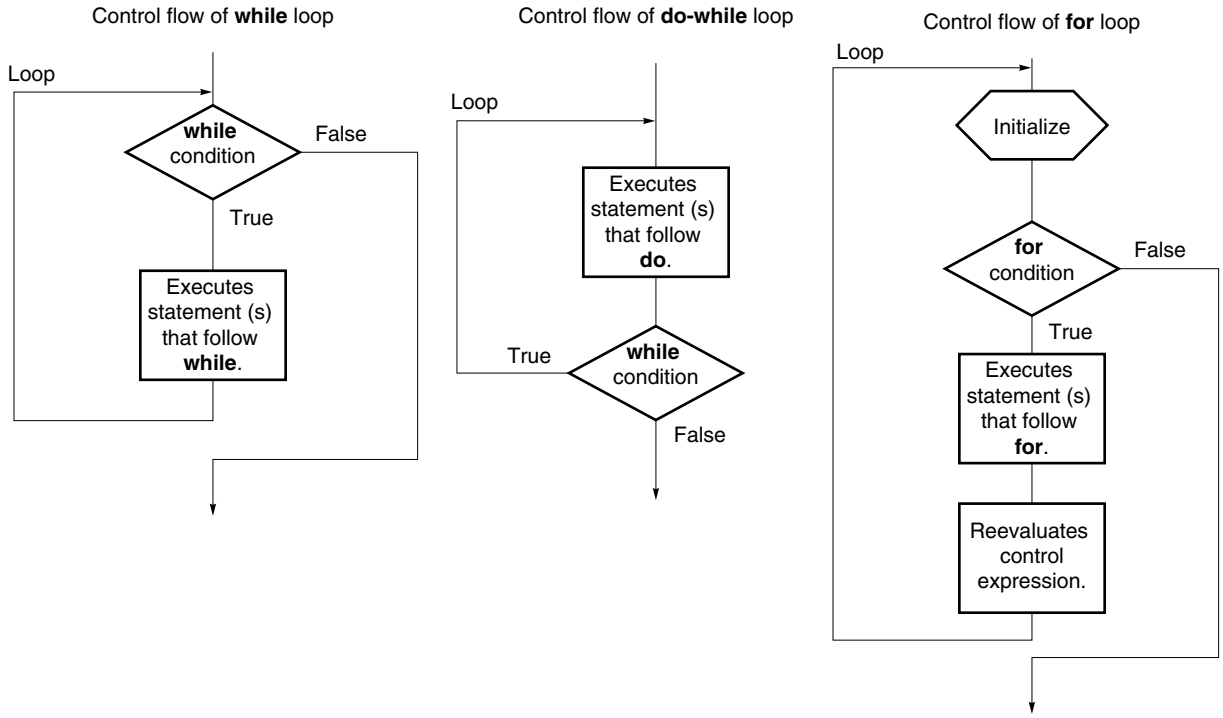
6.5 Iteration Statements

An 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 three types of iteration statements.

- `while` statement
- `do` statement
- `for` statement

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

Figure 6-2. Control Flows of Iteration Statements



(1) while statement

Iteration Statements — while 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.

SYNTAX

```
while (expression) statements
```

EXAMPLE

```
int   i, x ;
void main (void) {
     i=1, x=0 ;

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

EXPLANATION

The above example finds the sum total of integers from 1 to 10 for x. The two statements enclosed in braces 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.

(2) do statement

Iteration Statements — do statement

FUNCTION

A **do** statement executes the body of the loop as long as the control expression enclosed in parentheses is True (nonzero). The **do** statement evaluates the control expression after the loop body has been executed.

SYNTAX

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

EXAMPLE

```
Int  i, x ;
void main (void) {
    i=1, x=0 ;
    do {
      x += i ;
      i++ ;
    } while( i<11 );
}
```

EXPLANATION

The above example finds the sum total of integers from 1 to 10 for x. The two statements enclosed in braces 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.

(3) for statement

Iteration Statements — for statement

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 three 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 is executed 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 at the end of every loop and reevaluates the variable after the execution.

SYNTAX

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

EXAMPLE

```
int  i, x=0 ;

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

EXPLANATION

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.

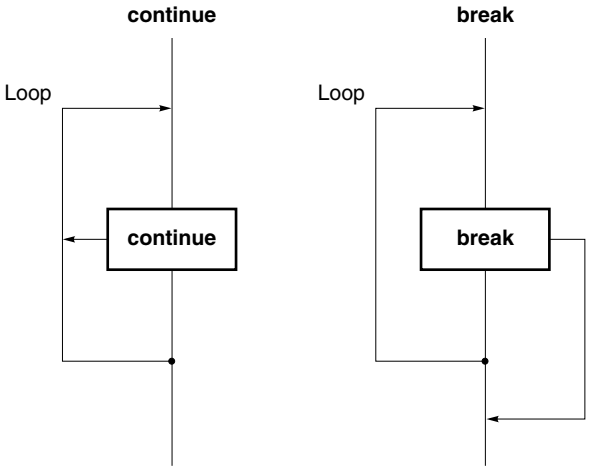
6.6 Branch Statements

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 four 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



(1) goto statement

Branch Statements goto

FUNCTION

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

SYNTAX

```
goto identifier ;
```

EXAMPLE

```
do {
        /*...*/
        goto point ;
        /*...*/
}while(/*...*/) ;
          /*...*/
point: ;
```

EXPLANATION

In the above example, when control is passed to the **goto** statement, C jumps out of the current **do ... while** loop processing unconditionally 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.

(2) continue statement

Branch Statements — 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 one loop, it ends the cycle of the smallest enclosing loop.

SYNTAX

```
continue ;
```

EXAMPLE

```
while(/*...*/){
    /*...*/
    continue ;
    /*....*/
    contin: ;
}
```

EXPLANATION

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.

(3) break statement

Branch Statements — 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.

SYNTAX

```
break ;
```

EXAMPLE

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

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

EXPLANATION

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.

(4) return statement

Branch Statements return

FUNCTION

A **return** statement exits the function that includes the return and passes controls to the function that called the return, and 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"}" at the end of a function produces the same result as when a **return** statement without expression is executed.

SYNTAX

```
return expression ;
```

EXAMPLE

```
Int  f(int);

void main(void) {
        /*...*/
        int  i = 0,  y = 0 ;
        y = f(i) ;
         /*...*/
}

int   (int  i) {
         int x = 0 ;
           /*...*/
         return(x) ;
}
```

EXPLANATION

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 with different types grouped under one 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 “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 the 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.

[Example]

```
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 two 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 two types. A bit field that specifies each member in bits can also be specified.

If a variable takes a binary value “0” or “1”, the minimum required number 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;
       unsigned int c:1;
} bit_field;
```

bit field

(3) Arrays and pointers

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

[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 (or structure element) may be referenced in two 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(){
       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(){
        char c;
        data_ptr -> tel [3] = 'E' ;
}
```

7.2 Unions

As mentioned earlier, a union is a collection of members that 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 “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” that have the 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. Individual elements in the union 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 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 two 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 two 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 declarations come after the preprocessing. These declarations are referred to as “external declarations” 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), one 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

main()
{
    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 forcibly converted by the compiler. This type conversion can be described by using the form of an identifier list for the parameters. An example of this identifier list is shown below.

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

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

```
int f(void);
void main( ){
     :
   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 two 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 an initializer. If the declaration of an identifier for an object that has file scope has no initializer without storage class specification or has **static** storage class, the object definition is considered to be temporary, because it becomes a declaration that has file scope with initializer 0.

Examples of external object definitions are shown below.

[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 PREPROCESSING DIRECTIVES (COMPILER DIRECTIVES)

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

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

A preprocessing directive specifies the processing performed before compiling a source file. Preprocessing directives include operations such as processing or skipping a part of a source file depending on the conditions, obtaining additional code from other source files, and replacing the original source code with other text as in macro expansion. The preprocessing directives are described below.

9.1 Conditional Translation Directives

Conditional translation 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 translation directive is 0, the statements that follow the directive are not translated (compiled). The **sizeof** operator, **cast** operator, or an enumerated type constant cannot be used in the constant expression of any conditional translation directive.

Conditional translation directives include **#if, #elif, #ifdef, #ifndef, #else,** and **#endif**.

In preprocessing directives, the following unary expressions called defined expressions may be used.

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

The unary expression returns 1 if the identifier has been defined with the **#define** preprocessing 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 compiles between **#if** and **#endif** because SYM has been defined (for the explanation of **#if** through **#endif**, refer to the explanations on the following pages).

```
#define SYM 0

#if defined SYM
    :
#endif
```

(1) #if directive

Conditional Translation Directives #if

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.

SYNTAX

```
#if constant expression line feed 「group」
```

EXAMPLE

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

EXPLANATION

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.

(2) #elif directive

Conditional Translation Directives #elif

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**.

SYNTAX

```
#elif constant-expression line feed ⌈group⌋
```

EXAMPLE

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

EXPLANATION

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.

(3) #ifdef directive

Conditional Translation Directives #ifdef

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**.

SYNTAX

```
#ifdef identifier line feed ⌈group⌋
```

EXAMPLE

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

EXPLANATION

In the above example, the identifier “ON” has been defined with the **#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.

(4) #ifndef directive

Conditional Translation Directives #ifndef

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.

SYNTAX

```
#ifndef identifier line feed ⌈group⌋
```

EXAMPLE

```
#define ON
#ifndef ON
        :
#endif
```

EXPLANATION

In the above example, the identifier “ON” has been defined with the **#define** directive. Thus, the source code between **#ifndef** and **#endif** will be discarded in the translation phase. If the identifier “ON” has never been defined, the source code between **#ifndef** and **#endif** will be translated.

(5) #else directive

Conditional Translation Directives #else

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.

SYNTAX

```
#else line feed [group]
```

EXAMPLE

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

EXPLANATION

In the above example, the identifier “ON” has been defined with the **#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.

(6) #endif directive

Conditional Translation Directives #endif

FUNCTION

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

SYNTAX

```
#endif line feed
```

EXAMPLE

```
#define ON
#ifdef ON
      :
      :

#endif
```

EXPLANATION

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

9.2 Source File Inclusion Directive

The preprocessing directive **#include** searches for a specified header file and replaces **#include** by the entire contents of the specified file. The **#include** directive may take one of the following three forms for inclusion of other source files.

- **#include** <file-name>
- **#include** “file-name”
- **#include** preprocessing token string

An **#include** directive may appear in the source obtained by **#include**. In this compiler, however, there are restrictions for **#include** directive nesting. For the restrictions, refer to **Table 1-1 Maximum Performance Characteristics of This C Compiler**.

Remark Preprocessing token string: Character string defined by the **#define** directive

(1) #include < >

Source File Inclusion Directive #include< >

FUNCTION

If the directive form is **#include** < >, the C compiler searches the directory specified by the **-i** compiler option, the directory specified by the INC78K environment variable, and the directory \NECTools32\INC78K4 registered in the registry for the header file specified in angle brackets and replaces the **#include** directive line with the entire contents of the specified file.

SYNTAX

```
#include <file-name> line feed
```

EXAMPLE

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

EXPLANATION

In the above example, the C compiler searches the directory specified by the INC78K environment variable and the directory \NECTools32\INC78K4 registered in the registry for the file **stdio.h** and replaces the directive line **#include<stdio.h>** with the entire contents of the specified file **stdio.h**.

Caution The above directories differ depending on the installation method.

(2) #include " "

Source File Inclusion Directive #include " "

FUNCTION

If the directive form is **#include** " ", the current working directory is first searched for the header file specified in double quotes. If it is not found, the directory specified by the **-i** compiler option, the directory specified by the INC78K environment variable, and the directory \NECTools32\INC78K4 registered in the registry are searched. The compiler then replaces the **#include** directive line with the entire contents of the specified file.

SYNTAX

```
#include "file-name" line feed
```

EXAMPLE

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

EXPLANATION

In the above example, the C compiler searches the current working directory, the directory specified by the INC78K environment variable, and the directory \NECTools32\INC78K4 registered in the registry for the file **myprog.h** specified in double quotes and replaces the directive line **#include "myprog.h"** with the entire contents of the specified file **myprog.h**.

Caution The above directories differ depending on the installation method.

(3) #include preprocessing token string

Source File Inclusion Directive #include token string

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.

SYNTAX

```
#include preprocessing token string  line feed
```

EXAMPLE

```
#define   INCFILE  "myprog.h"
#include  INCFILE
```

EXPLANATION

When including source files using the directive form "**#include** preprocessing token string line feed", the specified "preprocessing token string" must be substituted with <file-name> or "file name" by macro replacement. If the token string is replaced by <file-name>, the C compiler searches the directory specified by the **-i** compiler option, the directory specified by the INC78K environment variable, and the directory \NECTools32\INC78K4 registered in the registry for the specified file. If the token string is replaced by "file name", the current working directory is searched. If the specified file is not found, the directory specified by the **-i** compiler option, the directory specified by the INC78K environment variable, and the directory \NECTools32\INC78K4 registered in the registry are searched.

Caution The above directories differ depending on the installation method.

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 two forms: Object format and Function format.

- Object format
  `#define` identifier replacement list line feed

- Function format
  `#define` identifier (⌈identifier-list⌋) replacement-list line feed

(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 two tokens on either side of the **##** symbol into one 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() {
      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 replaced (not including the remaining preprocessing tokens in the source file) 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.

(6) #define directive

Macro Replacement Directives #define

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 line feed) whenever the same identifier appears in the source code after the definition by this directive.

SYNTAX

```
#define  identifier  replacement  list  line feed
```

EXAMPLE

```
#define  PAI  3.1415
```

EXPLANATION

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.

(7) #define( ) directive

Macro Replacement Directives #define ( )

FUNCTION

The function-form **#define** directive which 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 line feed). No white space is allowed between the first name and the opening parenthesis "(". This list of names (identifier list) may be empty. Because this form of the directive defines a macro, the macro call will be replaced with the parameters of the macro inside the parentheses.

SYNTAX

```
#define identifier ( ⌈identifier list⌋ ) replacement-list line feed
```

EXAMPLE

```
#define  F(n)  (n*n)
void main() {
int i;
i=F(2)
}
```

EXPLANATION

In the above example, **#define** directive will replace "F(2)" with "(2*2)" and thus the value of i will become 4. For the sake of safety, 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.

(8) #undef directive

Macro Replacement Directives #undef

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.

SYNTAX

```
#undef identifier line feed
```

EXAMPLE

```
#define F(n) (n*n)
     :
#undef F
```

EXPLANATION

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 preprocessing 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.

(1) To change the line number

To change the line number, the specification is made as follows. 0 and numbers larger than 32767 cannot be specified.

```
#line numeric-string line feed
```

[Example]

```
#line 10
```

(2) 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” line feed
```

[Example]

```
#line 10 “file1.c”
```

(3) To change using preprocessing token string

In addition to the specifications above, the following specification can also be made. In this case, the specified preprocessing token string must be either one of the above two examples after all the replacement.

```
#line preprocessing-token-string line feed
```

[Example]

```
#define  LINE_NUM   100
#line   LINE_NUM
```

9.5 #error Preprocessing Directive

The #error preprocessing directive is a directive that outputs a message including the specified preprocessing tokens and incompletely terminates compileation. This preprocessing directive is used to terminate compilation.

This preprocessing directive is specified as follows.

```
#error “preprocessing-token-string” line feed
```

[Example]

In this example, the macro name _ _**K4** _ _, which indicates the device series of this compiler, is used. If the device is the 78K/IV Series, the program between **#if** and **#else** is compiled. In the other cases, the program between **#else** and **#endif** is compiled, but compilation will be terminated with an error message “not for 78K4” output by the **#error** directive.

```
#if _ _K4_ _
     :
#else
#error   “not for 78K4”
  :
#endif
```

9.6 #pragma Directives

#pragma directives are directives to instruct the compiler to operate using the compiler definition method. In this compiler, there are several **#pragma** directives to generate codes for the 78K/IV Series (for details of **#pragma** directives, refer to **CHAPTER 11 EXTENDED FUNCTIONS)**.

[Example]

In this example, the **#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.

```
# line feed
```

9.8 Compiler-Defined Macro Names

In this C compiler, the following macro names have been defined.

_ _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 “Mmm 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** preprocessing directive must not be applied to these macro name and **defined** identifiers. All the macro names of the compiler definition start with an underscore followed by an uppercase character or a second underscore.

In addition to the above macro names, macro names indicating the series name of devices according to 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 or by the processor type in the C source.

- Macro name indicating the series name of devices
  ‘_ _K4 _ _’
- Macro name indicating the device name
  ‘_ _’ is added before the device type name and ‘_’ is added after the device type name.
  Describe letters in uppercase
  (Example) _ _4026_ _ _4038Y_

Remark The device type names are the same as those specified by the **-C** option. For the device type names, refer to the reference materials related to device files.

This C compiler has a macro name indicating the memory model or location.

- Macro name indicating memory model

  When small model is specified

  ```
  #define _ _K4_SMALL_ _          1
  ```

  When medium model is specified

  ```
  #define _ _K4_MEDIUM_ _         1
  ```

  When large model is specified

  ```
  #define _ _K4_LARGE_ _          1
  ```

- Macro name indicating location

  Location 0

  ```
  #define _ _K4LOC0_ _            1
  ```

  Location 15

  ```
  #define _ _K4LOC15_ _           1
  ```

The device type for compilation is specified by adding the following to the command line

'-c device type name'

Example `cc78k4 -c4038Y prime.c`

It is possible to avoid specifying the device type at compilation by specifying it at the start of the C source program.

'#pragma PC (device type)'

Example

```
#pragma PC (4038Y)
        :
```

However, the following can be described before '#pragma PC (device type)'

- Comment statement
- Preprocessing 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, C is provided with library functions to perform I/O operations.

This C compiler is provided with library functions such as I/O, character/memory manipulation, program control, and mathematical functions. This chapter describes the library functions provided in this compiler.

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 via a stack and a register, respectively. However, when the old function interface supporting option (**-ZO**) is not specified, the first argument is, if possible, also passed via a register.

For the **-ZO** option, refer to **CHAPTER 5 COMPILER OPTION** in the **CC78K4 C Compiler Operation User's Manual (U15557E)**.

10.1.1 Arguments

Placing or removing arguments on or from the stack is performed by the caller (calling function). The callee (called function) only references the argument values. However, when the argument is passed via a register, the callee directly refers to the register and copies the value of the argument to another register, if necessary. Also, when specifying the function call interface automatic pascal function option **-ZR**, removal of arguments from the stack is performed by the called side if the argument is passed by the stack.

Arguments are placed on the stack one by one in descending order from bottom to top if the argument is passed via the stack.

The minimum unit of data that 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.

When it is a large model and the argument is the address value or when it is a medium model and the argument is the address value of the function, the argument is stacked in 3-byte units.

The following shows the list of the passing of the first argument. The second and subsequent arguments are passed via a stack.

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 (Without -ZO Specification)	Passing Method (with -ZO Specification)
1-byte, 2-byte integer	AX	Passed via a stack
3-byte integer	WHL, small model: stack passing	Passed via a stack
4-byte integer	AX, RP2	Passed via a stack
Floating-point number (float type)	AX, RP2	Passed via a stack
Floating-point number (double type)	AX, RP2	Passed via a stack
Other	Passed via a stack	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 RPZ. When returning a structure, the first address of the structure is stored in the register BC. When returning a pointer, the first address of the structure is stored in the register BC.

The following shows the list of storing the return value. The method of storing return values is the same as that of normal functions.

Table 10-2. List of Storing Return Value

Return Value Type	Small Model	Medium Model	Large Model
1 bit	CY	CY	CY
1-byte, 2-byte integers	BC	BC	BC
4-byte integers	BC (lower), RP2 (higher)	BC (lower), RP2 (higher)	BC (lower), RP2 (higher)
Floating-point number (float type)	BC (lower), RP2 (higher)	BC (lower), RP2 (higher)	BC (lower), RP2 (higher)
Floating-point number (double type)	BC (lower), RP2 (higher)	BC (lower), RP2 (higher)	BC (lower), RP2 (higher)
Structure	Copies the structure to return to the area specific to the function and stores the address in BC	Copies the structure to return to the area specific to the function and stores the address in BC	Copies the structure to return to the area specific to the function and stores the address in TDE
Pointer	BC	BC (function pointer) WHL (function pointer)	TDE

10.1.3 Saving registers to be used by individual libraries

Each library that uses RP3, RG4 (VVP) and RG5 (UUP) 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.

(1) When -ZR option is not specified

The procedure of passing arguments and return values is shown below. An example of the small model is shown below.

Called function "`long func (int a, long b, char *c) ;`"

<1> Placing arguments on the stack (by the caller)
The higher 16 bits of arguments “c” and “b” and lower 16 bits of argument “b” are placed on the stack in the order named. a is passed via the AX register.

<2> Calling **func** by call instruction (by the caller)
The return address is placed on the stack next to the lower 16 bits of argument “b” and control is transferred to the function **func**.

<3> Saving registers to be used within the function (by the callee)
If register RP3 is to be used, RP3 is placed on the stack.

<4> Placing the first argument passed via the register on the stack (by the callee)

<5> Processing **func** and storing the return value in registers (by the callee)
The lower 16 bits of the return value “long” are stored in BC and the higher 16 bits of the return value, are stored are stored in RP2.

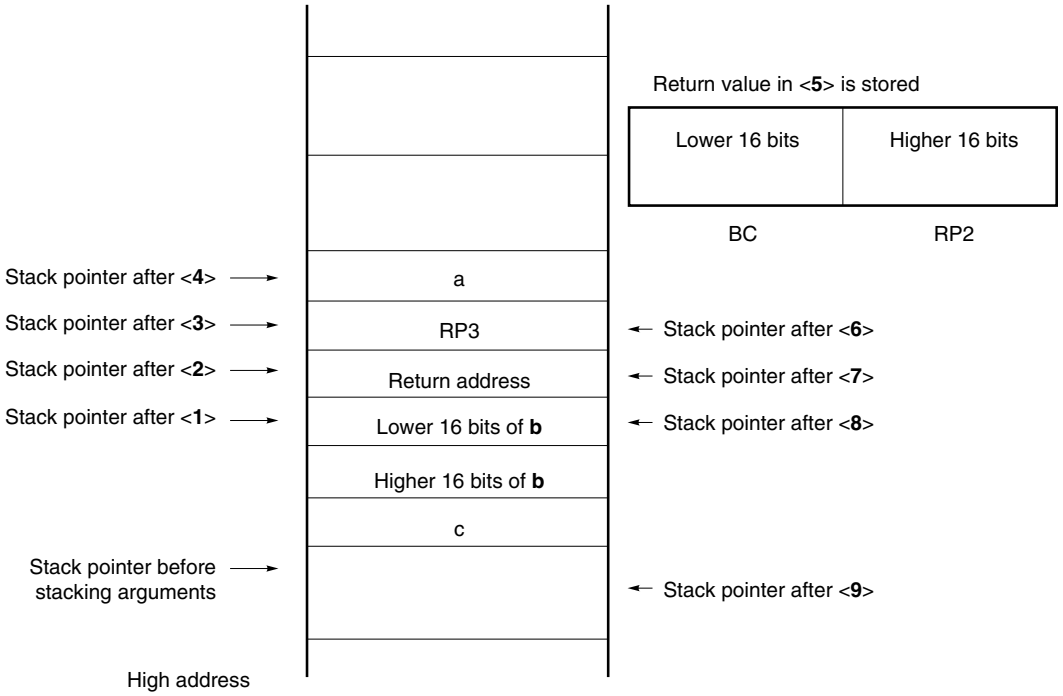
<6> Restoring the stored first argument (by the callee)

<7> Restoring the saved registers (by the callee)

<8> Returning control to the caller with **ret** instruction (by the callee)

<9> 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. In the example shown in Figure 10-1, 6 is added.

Figure 10-1. Stack Area When Function Is Called (No –ZR Specified)



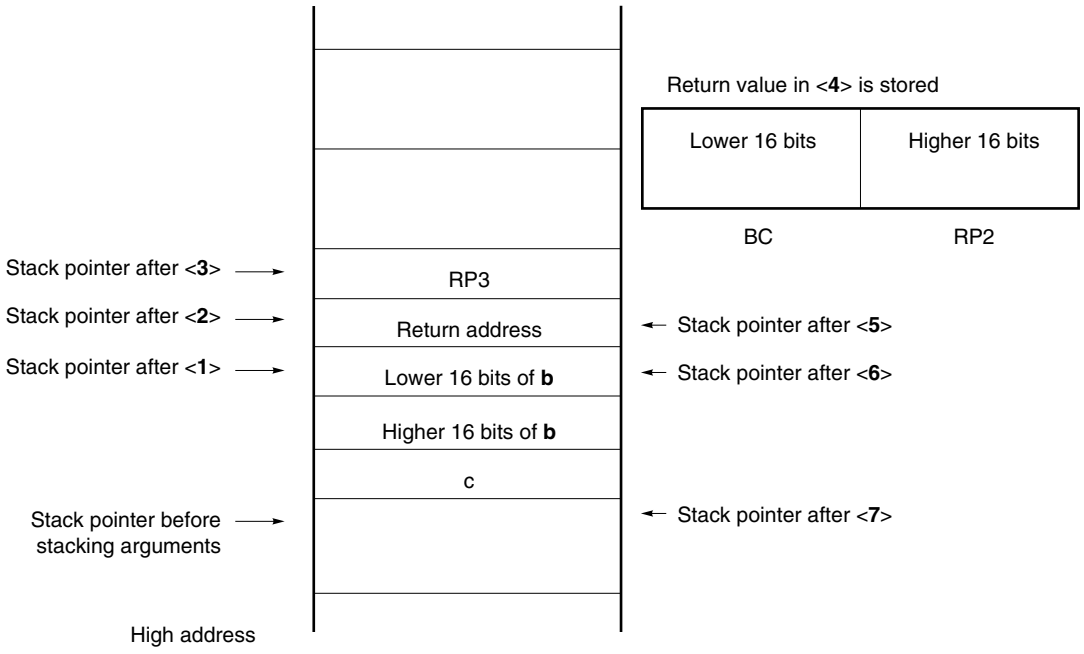
(2) When -ZR option is specified

The following example shows the procedure of passing arguments and return values when the **-ZR** option is specified.

Called function "`long func (int a, long b, char *c);`"

<1> Place arguments on the stack (by the caller)
The higher 16 bits of arguments "c" and "b" and the lower 16 bits of argument "b" are placed on the stack in the order named. a is passed via the AX register.
<2> Call **func** by call instruction (by the caller).
The return address is placed on the stack next to the lower 16 bits of argument "b" and control is transferred to the function **func**.
<3> Save the registers used in the functions (by the caller).
<4> Perform processing of the function **func**, and store return values in the register (by the callee).
Store the lower 16 bits of the return value (long) in BC and the higher 16 bits in RP2.
<5> Restore the saved registers (by the callee).
<6> Save the return address in the register (by the callee).
Save the return address in the WHL register.
<7> The caller restores the placed arguments (by the callee).
<8> Return control to the function on the caller in the branch instruction (by the callee) at the value saved in the register in <6>.
Return control to the function on the caller in the BR WHL instruction (by the callee).

Figure 10-2. Stack Area When Function Is Called (–ZR Specified)



10.2 Headers

This C compiler has 13 headers (or header files). Each header defines or declares standard library functions, data type names, and macro names.

These 13 headers 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 the compiler option **-ZA** (the option that disables the functions not complying with 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 **-ZA** is not specified, **tolow** and **toup** are not defined.

Isalnum	isalpha	iscntrl	isdigit	isgraph
islower	isprint	ispunct	isspace	isupper
isxdigit	tolower	toupper	isascii	toascii
_toupper	_tolower	tolow	toup	

(2) setjmp.h

This header is used to define program control functions. In this standard header, the **setjmp** and **longjmp** functions have been defined.

In the header **setjmp.h**, the following object has been defined.

[Declaration of **char** array type **jmp_buf** with an array size of greater than 30]

```
typedef char jmp_buf[30]
```

(3) stdarg.h

This header used to define special functions. In this standard header, the following four library functions have been defined.

When the **-ZO** option (old function interface supporting option) is not specified, the **va_start** function cannot be specified for the first argument because the first argument is passed via the register.

Use the macro in the following manner when the **-ZO** option is not specified.

- Use the **va_starttop** macro when specifying the first argument.
- Use the **va_start** macro when specifying the second argument.

```
va_start   va_starttop    va_arg    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 standard header, the following functions have been defined.

```
sprintf   sscanf    printf    scanf     vprintf   vsprintf
getchar   gets      putchar   puts
```

The following macro names are declared.

```
#define   EOF       (-1)
#define   NULL      (void *)0
```

(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 the compiler option **-ZA** (the option that disables the functions not complying with 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 **-ZA** is not specified, **strbrk**, **strsbrk**, **stritoa**, **strltoa**, and **strultoa** 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 ;
```

[Declaration of structure type **ldiv_t** which has **long int** type members **quot** and **rem**]

```
typedef struct {
        long int quot ;
        long int rem ;
} ldiv_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 standard header, the following library functions have been defined.

```
Memcpy  memmove  strcpy  stmcpy  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 standard 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 standard 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
```

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 standard 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))
```

• **offsetof** (type, member specifier)

offsetof is expanded to a general integer constant expression with the 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 the 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   tqnh   exp     frexp
ldexp  log    log10  modif  pow    sqrt   ceil   fabs   floor  fmod   acosf
asinf  atanf  atan21 cost   sinf   tanf   coshf  sinhf  tanhf  expf   frexpf
ldexpf logf   log10f modff  powf   sqrtf  ceilf  fabsf  floorf fmodf  matherr
```

The following objects are defined.

[Definition of macro name **HUGE_VAL**]

```
#define HUGE_VAL _HUGE
```

(12) float.h

float.h defines the following objects.

When the size of a **double** type is 32 bits, the macros 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                 1.1749435E-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 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, the **assert.h** header file is not defined in the **assert.h** header file.

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 **assert.h** is captured to the source file, the **assert.h** header file simply declares the assert macro as:

```
#define assert(p) ((void)0)
```

and does not define _ _ **assertfail**.

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 this compiler does not use static area allowing re-entrantability. Therefore, data in the storage used by functions will not be destroyed by a call from another program.

However, the functions shown in (1) to (3) are not re-entrant.

(1) Functions that cannot be re-entranced

setjmp, **longjmp**, **atexit**, **exit**

(2) Functions that use the area secured in the startup routine

div, **ldiv**, **brk**, **sbrk**, **rand**, **srand**, **strtok**

(3) Functions that deal with floating-point numbers

sprintf, **sscanf**, **printf**, **scanf**, **vprintf**, **vsprintf** [Note], **atof**, **strtod**, and all the mathematical functions

Note Among **sprintf**, **sscanf**, **printf**, **scanf**, **vprintf**, **and vsprintf**, functions that do not support floating-point numbers are re-entrant.

10.4 Standard Library Functions

This section explains the standard library functions of this C compiler classified by function as follows. All standard library functions are supported even when the **–ZF** option is specified.

- Item (1-x) Character and character string functions
- Item (2-x) Program control functions
- Item (3-x) Special functions
- Item (4-x) I/O functions
- Item (5-x) Utility functions
- Item (6-x) Character string/memory functions
- Item (7-x) Mathematical functions
- Item (8-x) Diagnostic functions

1-1 is~ Character & String Functions

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	1 if character c is included in the character range. 0 if character c is not included in the character range.

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, line feed, vertical tab, and form feed)
ispunct	Punctuation characters except white-space characters
isprint	Printable characters
isgraph	Printable nonblank characters
iscntrl	Control characters
isascii	ASCII character set

1-2 toupper, tolower — Character & String Functions

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  to~(int c);
```

Function	Arguments	Return Value
toupper, tolower	c… Character to be converted	Uppercase equivalent if c is a convertible character. Character “c” is returned unchanged if not convertible.

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.

1-3 toascii Character & String Functions

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.

1-4 _toupper/toup _tolower/tolow — Character & String Functions

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 **tolow**)

Remark a: Lowercase, A: Uppercase

HEADER

ctype.h

FUNCTION PROTOTYPE

```
int _to~(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 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.

2-1 setjmp, longjmp — Program Control Functions

FUNCTION

The program control function **setjmp** saves the environment information 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);
```

(**jmp_buf** is **typedef** defined with **setjmp.h**.)

Function	Arguments	Return Value
setjmp	**env** ... Array to which environment information is to be saved	• 0 if called directly • Value given by “val” if returning from the corresponding **longjmp** 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** function saves the RP3, RG4, RG5 registers, **saddr** area and **SP** to be used as variable registers, and the return address of the functions to the array (or information block) referred to as **env** and returns 0.

longjmp

- The **longjmp** function restores the environment information (RP3, RG4, RG5 registers, **saddr** area and **SP** to be used as variable registers) saved to **env**. Program execution continues as if the value given by **val** (or 1 if the value of **val** is 0) was returned by the corresponding **setjmp**.

3-1 va_start, va_starttop, va_arg, va_end — Special Functions

FUNCTION

The **va_start** function (macro) is used to start a variable argument list.

The **va_starttop** function (macro) is used to start a variable argument list.

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);
```

va-list is typedef defined with stdarg.h.

Function	Arguments	Return Value
va_start **va_starttop**	**va_list** ..... Variable argument list **ap** ... Variable to be initialized so that it can be used in **va_arg** and **va_end** **parmN** ... Name of last parameter in function prototype (one immediately proceeding ellipsis "...")	None
va_arg	**va_list ap** ... Variable argument list. **ap** must be set up with call to **va_start** before calling **va_arg** type... Type of argument to be obtained	Next value from argument list; 0 if ap is a null pointer
va_end	**va_list ap** .... Variable argument list. **ap** must be set up with call to **va_start** before calling **va_arg**.	None

va_start, va_starttop, va_arg, va_end — Special Functions

EXPLANATION

va_start

- In the **va_start** macro, the argument **ap** (argument pointer) 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 (rightmost) 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, proper operation of this function is not guaranteed.

va_starttop

- When the **-ZO** option (old function interface supporting option) is not specified, the **va_start** function cannot be specified for the first argument because the first argument is passed via the register.
  Use the macro in the following manner when the **-ZO** option is not specified.
- Use the **va_starttop** macro when specifying the first argument.
- Use the **va_start** macro when specifying the second argument.

va_arg

- In the **va_arg** macro, the argument **ap** must be the same as the **va_list** type object initialized with **va_start**.
- After the argument pointer **ap** has been initialized via a call to **va_start**, parameters are returned via calls to **va_arg**, with **type** being the type of the next parameter. (Each call to **va_arg** obtains the next value from the argument list.)
- If the argument pointer **ap** is a null pointer, 0 (of **type** type) is returned.

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.

4-1 sprintf I/O Functions

FUNCTION

sprintf writes data into a character string 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 that 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. If the formats run out despite the fact that 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:
  - Zero or more flags (to be explained later) that modify the meaning of the format command
  - Optional decimal integer that specifies a minimum field width
    If the output width after the conversion is less than this minimum field width, this specifier pads the output with spaces or 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 if the minimum is exceeded.
  - 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 the **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.

sprintf I/O Functions

- 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.

– ..... 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, a 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.

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.

sprintf I/O Functions

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 one decimal number, and dddd is one or more decimal number(s). ddd is exactly a three-digit decimal number, and the sign is + or –. When the precision is omitted, it is interpreted as 6

E ..... The same format as that of e except E is added instead of e before the exponent.

g ..... Uses whichever shorter method of f or e format when converting double argument based on the specified precision. e format is used only when the exponent of the value is smaller than –4 or larger than the specified number by precision.
The following 0 are truncated, and the decimal point is displayed only when one or more numbers follow.

G ..... The same format as that of g except E is added instead of e before the exponent.

c ..... Converts int argument to unsigned char and writes the result as a single character.

s ..... The associated argument is a pointer to a string of characters and the characters in the string are written up to the terminating null character (but not included in the output). If precision is specified, the characters exceeding the maximum field width will be truncated off the end. When the precision is not specified or larger than the array, the array must include a null character.

p ..... The associated argument is a pointer to **void** and the pointer value is displayed in unsigned hexadecimal 4 digits (with 0s prefixed to less than a 4-digit pointer value). In the case of the large model, the pointer value is displayed in unsigned hexadecimal 8 digits (the higher 2 digits are padded by 0 and displayed with 0s prefixed to less than a 6-digit pointer value). The precision specification if any will be ignored.

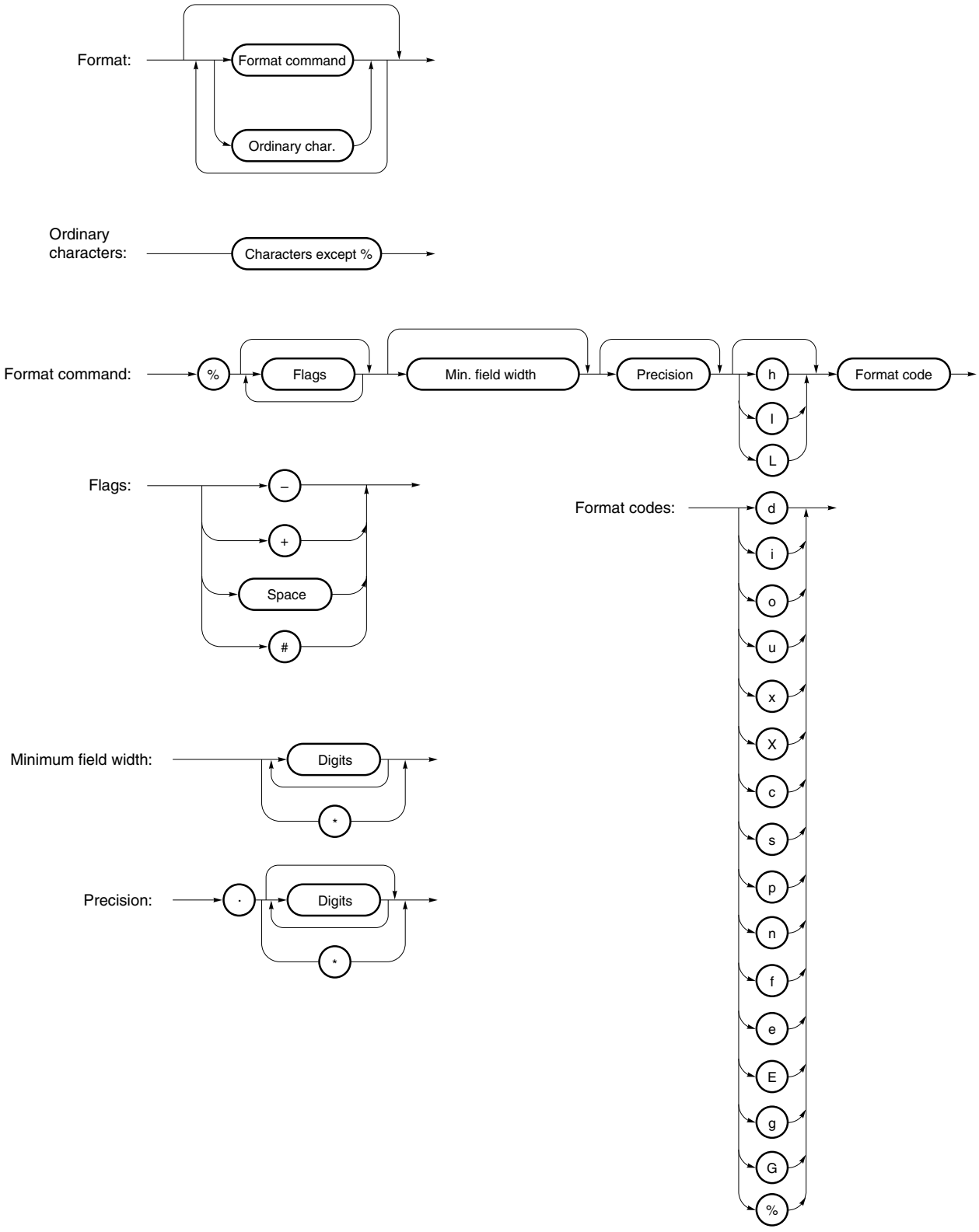
n ..... The associated argument is an integer pointer into which the number of characters written thus far in the string "s" is placed. No conversion is performed.

% ..... Prints a % sign. The associated argument is not converted (but the flag and minimum field width specifications are effective).

- Operations for invalid conversion specifiers are not guaranteed.
- When the actual argument is a union or a structure, or the pointer to indicate them (except the character type array in % s conversion or the pointer in % p conversion), operations are not guaranteed.
- The conversion result will not be truncated even when there is no field width or the field width is small. In other words, when the number of characters of the conversion result are larger than the field width, the field is extended to the width that includes the conversion result.
- The formats of the special output character string in %f, %e, %E, %g, %G conversions are shown below.
  non-numeric → "(NaN)"
  +∞ → "(+INF)"
  –∞ →"(–INF)"

sprintf writes a null character at the end of the string **s**. (This character is included in the return value count.)
The syntax of **format** commands is illustrated in Figure 10-3.

Figure 10-3. Syntax of Format Commands



4-2 sscanf I/O Functions

FUNCTION

sscanf reads data from the input string 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 that indicates the input format commands **...** ... Pointer to object in which converted values are to be stored, and zero or more arguments	–1 if the string **s** is empty. Number of assigned input data items if the string **s** is not empty.

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 classified into the following three 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:
  - Optional * character which suppresses assignment of data to the corresponding argument
  - Optional decimal integer which specifies a maximum field width
  - 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 described later)

sscanf I/O Functions

- **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 characters 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 characters.
  (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.
    - The input white-space characters (specified by **isspace**) are skipped over, except when the type specifier is **[, c,** or **n**.
    - 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.
    - 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 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 that follows “format” and has not yet received the result of the conversion.
- The following type specifiers are available.

d..... Reads a decimal integer (which may be signed). The corresponding argument must be a pointer to an integer.

i..... Reads 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..... Reads an octal integer (which may be signed). The corresponding argument must be a pointer to an integer.

u..... Reads an unsigned decimal integer.
The corresponding argument must be a pointer to an unsigned integer.

x..... Reads a hexadecimal integer (which may be signed).

e, **E**, **F**, **g**, **G**..... A floating-point value consists of an optional sign (+ or –), one or more consecutive decimal number(s) including a decimal point, an 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 ±∞, a non-normalized number or ±0 becomes the conversion result. The corresponding argument is a pointer to **float**. 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.

sscanf I/O Functions

s ..... Inputs a character string consisting of a non-blank character string. The corresponding argument is a pointer to an integer. 0x or 0X can be allocated at the first hexadecimal integer. The corresponding argument must be a pointer an array that has sufficient size to accommodate this character string and a null terminator. The null terminator will be automatically added.

[..... Inputs a character string consisting of expected character groups (called a **scanset**). The corresponding argument must be a pointer to the first character of an array that has sufficient size to accommodate this character string and a null terminator. The null terminator will be automatically added. The format commands continue from this character up to the closing square bracket (]). The character string (called a **scanlist**) enclosed in the square brackets constitutes a **scanset** except when the character immediately after the opening square bracket is a circumflex (^).

When the character is a circumflex, all the characters other than a **scanlist** between the circumflex and the closing square bracket constitute a **scanset**. However, when a **scanlist** begins with [ ] or [^], this closing square bracket is contained in the **scanlist** and the next closing square brocket 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. For the large model, a hexadecimal 8-digit integer is input, and the higher two digits are ignored.

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.

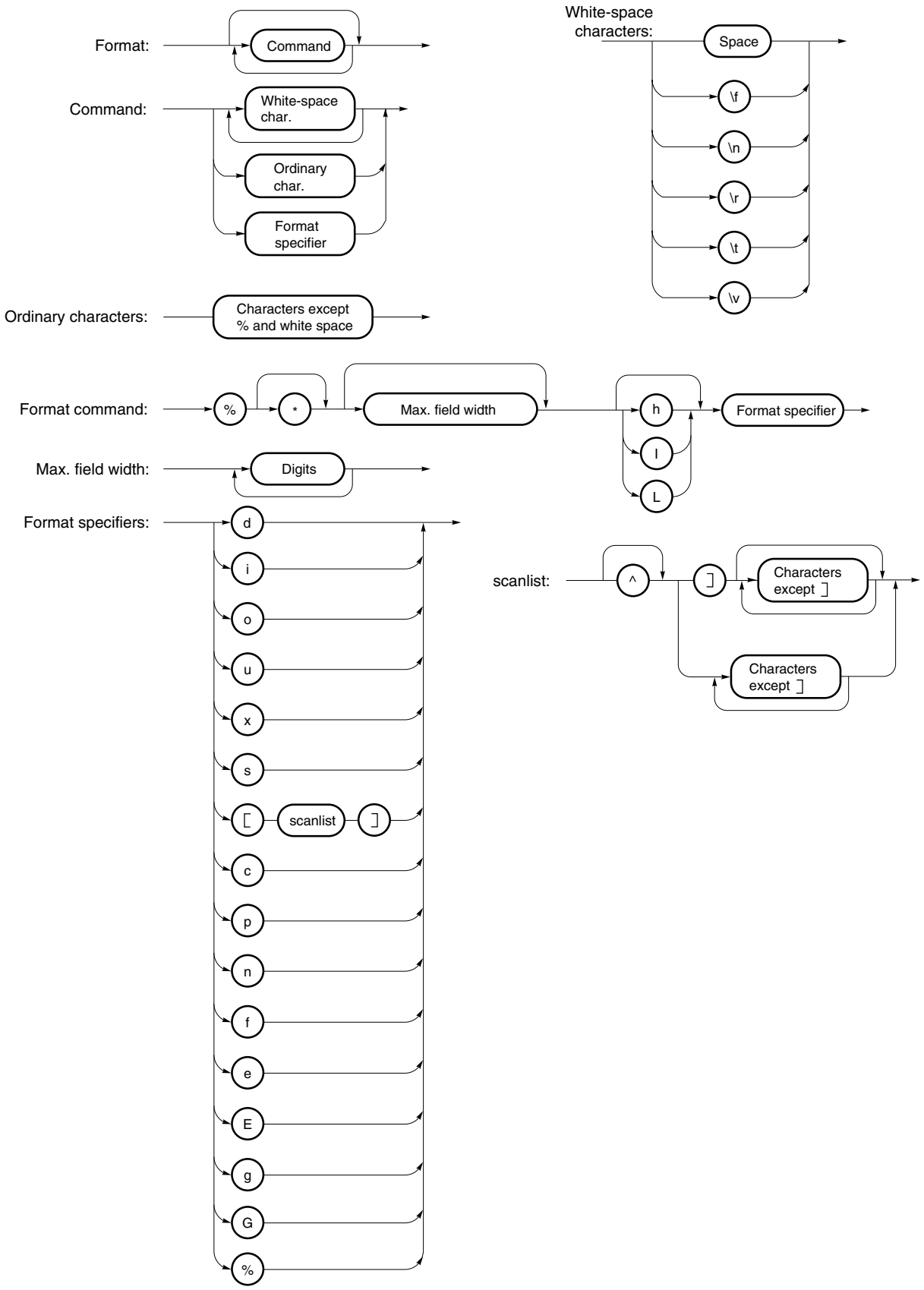
If a format specification is invalid, the format command execution fails.

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), the higher 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.

Figure 10-4. Syntax of Input Format Commands



4-3 printf I/O Functions

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 characters 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 conversion specifications 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.

4-4 scanf I/O Functions

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 format **...** ... 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 the **getchar** function. Specifies the input string permitted by the character string indicated by **format**. Uses the arguments after **format** as pointers to an object. **format** specifies how the conversion is performed by the input string.
- When there are not enough arguments for **format**, normal operation is not guaranteed. When the number of arguments 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 an input character that conflicts with the directive, the conflicting input character is rounded down. The conversion indication is the same as that of the **sscanf** function.

4-5 vprintf I/O Functions

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 the **putchar** function according to the output conversion specification specified by the format.
- Each conversion specification is the same as that of the **sprintf** function.

4-6 vsprintf I/O Functions

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 indicated by **s** according to the output conversion specification specified by **format**.
- The output specification is the same as that of the **sprintf** function.

4-7 getchar I/O Functions

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).
- An error check related to reading is not performed.
- To change the SFR to be read, it is necessary to either change the source and re-register it to the library or create a new **getchar** function.

4-8 gets I/O Functions

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 remain unchanged, and a null pointer is returned.

4-9 putchar I/O Functions

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).
- An error check related to writing is not performed.
- To change the SFR to be written, it is necessary to either change the source and re-register to the library or user create a new **putchar** function.

4-10 puts I/O Functions

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 the **putchar** function and adds a line feed character 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 the **putchar** function returns –1, –1 is returned.

5-1 atoi, atol — Utility Functions

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** 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	• **int** value if converted properly • **INT_MAX** (32767) if positive overflow occurs • **INT_MIN** (–32768) if negative overflow occurs • 0 if the string is invalid
atol		• **long int** value if converted properly • **LONG_MAX** (2147483647) for positive overflow • **LONG_MIN** (–2147483648) for negative overflow • 0 if the string is invalid

atoi, atol — Utility Functions

EXPLANATION

atoi

- The **atoi** function converts the first part of the string pointed to by pointer “nptr” to an **int** value. The string may consist of zero or more white-space characters possibly followed by a minus or plus sign, followed by a string of digits.
- 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 **int** value (until other than digits or a null character appears in the string).
- If no digits to convert are found in the string, the function returns 0. If an overflow occurs, the function returns **INT_MAX** (32767) for a positive overflow and **INT_MIN** (–32768) for a negative overflow.

atol

- The **atol** function converts the first part of the string pointed to by pointer “nptr” to a **long** value. The string may consist of zero or more white-space characters, possibly followed by a minus or plus sign, followed by a string of digits.
- 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 a **long** value (until other than digits or null character appears in the string).
- If no digits to convert are found in the string, the function returns 0. If an overflow occurs, the function returns **LONG_MAX** (2147483647) for a positive overflow and **LONG_MIN** (–2147483648) for a negative overflow.

5-2 strtol, strtoul Utility Functions

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	• **long int** value if converted properly • **LONG_MAX** (2147483647) for positive overflow • **LONG_MIN** (–2147483648) for negative overflow • 0 if not converted
strtoul		• **unsigned long** if converted properly • **ULONG_MAX** (4294967295U) if overflow occurs • 0 if not converted

strtol, strtoul — Utility Functions

EXPLANATION

strtol

- The **strtol** function decomposes the string pointed by pointer **nptr** into the following three parts.
  (1) String of white-space characters that may be empty (to be specified by **isspace**)
  (2) Integer representation by the base determined by the value of "base"
  (3) String of one or more characters that cannot be recognized (including null terminators)
  The **strtol** function converts part (2) of the string into a long 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 part (3) of the string is stored in the object pointed to by **endptr**.
- If the correct value causes an overflow, the function returns **LONG_MAX** (2147483647) for the positive overflow or **LONG_MIN** (–2147483648) for the negative overflow depending on the sign and sets **errno** to ERANGE (2).
- If the string in (2) is empty or the first non-white-space character of the string (2) is not appropriate for an integer with the given base, the function performs no conversion and returns 0. In this case, the value of the string **nptr** is stored in the object pointed to by **endptr** (if it is not a null string). This holds true with the bases 0 and 2 to 36.

strtoul

- The **strtoul** function decomposes the string pointed by pointer **nptr** into the following three parts.
  (1) String of white-space characters that may be empty (to be specified by **isspace**)
  (2) Integer representation by the base determined by the value of **base**
  (3) String of one or more characters that cannot be recognized (including null terminators)
  The **strtoul** function converts part (2) of the string into a unsigned long integer and returns this unsigned long 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 part (3) of the string is stored in the object pointed to by **endptr**.

strtol, strtoul — Utility Functions

- If the correct value causes an overflow, the function returns **ULONG_MAX** (4294967295U) and sets **errno** to **ERANGE** (2).
- If the string in (2) is empty or the first non-white-space character of the string in (2) is not appropriate for an integer with the given base, the function performs no conversion and returns 0. In this case, the value of the string **nptr** is stored in the object pointed to by **endptr** (if it is not a null string). This holds true with the bases 0 and 2 to 36.

5-3 calloc Utility Functions

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	• Pointer to the beginning of the allocated area if the requested size is allocated • Null pointer if the requested size is not allocated

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.
- 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 space will become a new break value. See **5-11 brk** for break value setting with the memory function **brk**.)

5-4 free Utility Functions

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**. (**malloc, calloc,** or **realloc** called after **free** will allocate space that was freed earlier.)
- If **ptr** does not point to the allocated space, **free** will take no action. (Freeing the allocated space is performed by setting **ptr** as a new break value.)

5-5 malloc Utility Functions

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	• Pointer to the beginning of the allocated area if the requested size is allocated • Null pointer if the requested size is not allocated

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. See **5-11 brk** for break value setting with the memory function **brk**.)

5-6 realloc Utility Functions

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	• Pointer to the beginning of the reallocated space if the requested size is reallocated • Pointer to the beginning of the allocated space if **ptr** is a null pointer • Null pointer if the requested size is not reallocated or “ptr” is not a 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.)

5-7 abort Utility Functions

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.

5-8 atexit, exit — Utility Functions

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	• 0 if function is registered as wrap-up function • 1 if function cannot be registered
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 wrap-up function can be registered, **atexit** returns 0. If no more wrap-up functions 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.

5-9 abs, labs — Utility Functions

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	• Absolute value of **j** if **j** falls within • –32767 ≤ j ≤ 32767 • –32768 (0x8000) if **j** is –32768
labs	**j** ... Any long integer for which absolute value is to be obtained	• Absolute value of **j** if **j** falls within –2147483647 ≤ j ≤ 2147483647 • –2147483648 (0x80000000) if the value of **j** is –2147483648

EXPLANATION

abs

- The **abs** returns the absolute value of its **int** type argument. If **j** is –32768, the function returns –32768.

labs

- The **labs** returns the absolute value of its **long** type argument. If the value of **j** is –2147483648, the function returns –2147483648.

5-10 div, ldiv — Utility Functions

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 of structure type **div_t** and the remainder to the **rem** element
ldiv		Quotient to the **quot** element of structure type **ldiv_t** and the remainder to the **rem** element

EXPLANATION

div

- The **div** function performs the integer division of numerator divided by denominator. The result of **div** has a structure type named **div_t** with the elements **quo** (quotient) and **rem** (remainder).
- The absolute value of the 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 –32768 and **denom** is –1, the quotient becomes –32768 and the remainder becomes 0.

ldiv

- The **ldiv** function performs the long integer division of numerator divided by denominator. The result of **ldiv** has a structure type named “ldiv_t” with the elements **quo** (quotient) and **rem** (remainder).
- The absolute value of the 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 –2147483648 and **denom** is –1, the quotient becomes –2147483648 and the remainder becomes 0.

5-11 brk, sbrk — Utility Functions

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	• 0 if break value is set properly • –1 if break value cannot be changed
sbrk	**incr** ... Value (bytes) by which set break value is to be incremented/decremented.	• Old break value if incremented or decremented properly • –1 if old break value cannot be incremented or decremented

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).

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 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).

5-12 atof
strtod — Utility Functions

FUNCTION

atof converts a decimal integer character string to **double**.

strtod converts a character string to **double**.

HEADER

stdlib.h

FUNCTION PROTOTYPE

```
double atof const char *nptr) ;
double strtod (const char *nptr, char **endptr) ;
```

Function	Arguments	Return value
atof	**nptr** ... Character string to be converted **endptr** ... Pointer to store a pointer to an unidentifiable area (**strtod** only)	• Normal ... Converted value • When positive overflow occurs ... **HUGE_VAL** (with the sign of the overflowed value) When negative overflow occurs ... 0 Illegal character string ... 0
strtod	**nptr** ... Character string to be converted **endptr** ... Pointer to store a pointer to an unidentifiable area	• Normal ... Converted value • When positive overflow occurs ... **HUGE_VAL** (with the sign of the overflowed value) When negative overflow occurs ... 0 Illegal character string ... 0

5-12 atof strtod — Utility Functions

EXPLANATION

atof

- **atof** converts the character string that is pointed by the pointer **nptr** to **double**.
- Skips 0 or more strings of null characters (a character which makes **isspace** true) from the start and converts the character string (other than decimal characters or until the last null character appears) from the character next to the floating-point number.
- If the conversion is performed correctly, a floating point number is returned.
- If an overflow occurs in the conversion, **HUGE_VAL**, which has the sign of the overflowed value, is returned, and **ERANGE** is set to **errno**.
- If annihilation of valid digits occurs due to underflow or overflow, a non-normalized number and ±0 are returned, respectively, and **ERANGE** is set to **errno**.
- If a conversion cannot be performed, 0 is returned.

strtod

- **strtod** converts the character string that is pointed by the pointer **nptr** to **double**.
- Skips 0 or more strings of null characters (a character which makes **isspace** true) from the start and converts the character string (other than decimal characters or until the last null character appears) from the character next to the floating-point number.
- If the conversion is performed correctly, a floating-point number is returned.
- If an overflow occurs in the conversion, **HUGE_VAL**, which has the sign of the overflowed value, is returned, and **ERANGE** is set to **errno**.
- If annihilation of valid digits occurs due to underflow or overflow, a non-normalized number and ±0 are returned, respectively, and **ERANGE** is set to **errno**. At the same time, **endptr** stores the pointer in the next character string.
- If conversion cannot be performed, 0 is returned.

5-13 itoa, ltoa, ultoa — Utility Functions

FUNCTION

The string function **itoa** converts an **int** integer to its string equivalent.

The string function **ltoa** converts a **long** 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	• Pointer to the converted string if converted properly • Null pointer if not converted properly

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.

5-14 rand, srand — Utility Functions

FUNCTION

The mathematical function **rand** generates a sequence of psuedorandom 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	Psuedorandom integer in the range of 0 to **RAND_MAX**
srand	**seed** ... Starting value for psuedorandom number generator	None

EXPLANATION

rand

- Each time the **rand** function is called, it returns a psuedorandom 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 psuedorandom 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.)

5-15 bsearch Utility Functions

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 that contains information to search **nmemb** ... Number of array elements **size** ... Size of an array **compare** ... Pointer to function used to compare two keys	• Pointer to the first member that matches “key” if the array contains the key • Null pointer if the key is not contained in the array

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 that 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 two arguments (**key** as the 1st argument and array element as the 2nd argument), compares the two 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
- When the **-ZR** option is specified, the function passed to the argument of the **bsearch** function must be a pascal function.

5-16 qsort Utility Functions

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 two 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 two arguments (array element 1 as the 1st argument and array element 2 as the 2nd argument), compares the two 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
- If the two array elements are equal, the element nearest to the top of the array will be sorted first.
- When the **-ZR** option is specified, the function passed to the argument of the **qsort** function must be a pascal function.

5-17 strbrk Utility Functions

FUNCTION

strbrk sets a break value.

HEADER

stdlib.h

FUNCTION PROTOTYPE

```
int strbrk (char *endds);
```

Function	Arguments	Return Value
strbrk	**endds** ... Break value to be 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.

5-18 strsbrk Utility Functions

FUNCTION

strsbrk increments/decrements a break value.

HEADER

stdlib.h

FUNCTION PROTOTYPE

```
char *strsbrk (int incr);
```

Function	Arguments	Return Value
strsbrk	**incr** ... Amount by which a break value is to be incremented/decremented	Normal ... Old break value When a break value cannot be incremented/decremened ... –1

EXPLANATION

- **incr** byte increments/decrements a break value (depending on the sign of incr).
- When the break value is out of the permissible range after incrementing/decrementing, the break value is not changed. **ENOMEM**(3) is set to errno, and –1 is returned.

5-19 stritoa strltoa strultoa — Utility Functions

FUNCTION

stritoa converts **int** to a character string.

strltoa converts **long** to a character string.

strultoa converts **unsigned long** to a character string.

HEADER

stdllib.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 Other ... Null pointer

EXPLANATION

stritoa, strltoa, strultoa

- Converts the specified numeric value **value** to the character string that ends with a null character, and stores the result in the area specified with **string**. The conversion is performed by the specified **radix**, and the pointer to the converted character string will be returned.
- **radix** must be a value in the range of 2 to 36. In other cases, the conversion is not performed and a null pointer is returned.

6-1 memcpy, memmove — Character String/Memory Functions

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.

6-2 strcpy, strncpy Character String/Memory Functions

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, **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 string pointed by **s2** is less than the characters specified by **n**, nulls will be appended to the end of **s1** until **n** characters have been copied. If the string pointed to by **s2** is longer than **n** characters, the resultant string that is pointed to by **s1** will not be null terminated.

6-3 strcat, strncat — Character String/Memory Functions

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, 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**.
- **s1** must always be terminated with a null.
- When copying is performed between objects overlapping each other, the operation is not guaranteed.

6-4 memcmp Character String/Memory Functions

FUNCTION

The memory function **memcmp** compares two 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 two data objects to be compared **n** ... Number of characters to compare	• 0 if **s1** and **s2** are equal • Positive value if **s1** is greater than **s2**; negative value if **s1** is less than **s2** (**s1** – **s2**)

EXPLANATION

- The **memcmp** function compares the data object pointed to by **s1** with the data object pointed to by **s2** with respect to the number of bytes specified by **n**.
- If the two objects are equal, the function returns 0.
- The function returns a positive value if the object **s1** is greater than the object **s2** and a negative value if **s1** is less than **s2**.

6-5 strcmp, strncmp — Character String/Memory Functions

FUNCTION

The string function **strcmp** compares two character strings.

The string function **strncmp** compares not more than a specified number of characters from two 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	• 0 if **s1** is equal to **s2** • Integer less than 0 or greater than 0 if **s1** is less than or greater than **s2** (**s1** – **s2**)
strncmp	**s1**... Pointer to one string to be compared **s2** ... Pointer to the other string to be compared **n** ... Number of characters to be compared	• 0 if **s1** is equal to **s2** within characters specified by **n** • Integer less than 0 or greater than 0 if **s1** is less than or greater than **s2** (**s1** – **s2**) within characters specified by **n**

EXPLANATION

strcmp

- The **strcmp** function compares the two null terminated strings pointed to by **s1** and **s2**, respectively.
- If **s1** is equal to **s2**, the function returns 0. If **s1** is less than or grater than **s2**, the function returns an integer less than 0 (a negative number) or greater than 0 (a positive number) (**s1** – **s2**).

strncmp

- The **strncmp** function compares not more than the characters specified by **n** from the two null terminated strings pointed to by **s1** and **s2**, respectively.
- If **s1** is equal to **s2** within the specified characters, the function returns 0. If **s1** is less than or greater than **s2** within the specified characters, the function returns an integer less than 0 (a negative number) or greater than 0 (a positive number) (**s1** – **s2**).

6-6 memchr Character String/Memory Functions

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	• Pointer to the first occurrence of **c** if **c** is found • Null pointer if **c** is not found

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.

6-7 strchr, strrchr Character String/Memory Functions

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	• Pointer indicating the first or last occurrence of **c** in string **s** if **c** is found in **s** • Null pointer if **c** is not found in **s**

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.

6-8 strspn, strcspn Character String/Memory Functions

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 *2);
```

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**.

6-9 strpbrk Character String/Memory Functions

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	• Pointer to the first character in the string **s1** that matches any character in the string **s2** if any match is found • Null pointer if no match is found

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.

6-10 strstr Character String/Memory Functions

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	• Pointer to the first appearance in the string **s1** of the string **s2** if **s2** is found in **s1** • Null pointer if **s2** is not found in **s1** • Value of **s1** if **s2** is a null string

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**.

6-11 strtok Character String/Memory Functions

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	• Pointer to the first character of a token if it is found • Null pointer if there is no token to return

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.

6-12 memset Character String/Memory Functions

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**.

6-13 strerror Character String/Memory Functions

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	• Pointer to string describing error message if message associated with error number exists • Null pointer if no message associated with error number exists

EXPLANATION

- The **strerror** function returns a pointer to one of the following strings associated with the value of **errnum** (error number):

  0 ..... “Error 0”
  1 (EDOM) ..... “Argument too large”
  2 (ERANGE)..... “Result too large”
  3 (ENOMEM)..... “Not enough memory”

  Otherwise, the function returns a null pointer.

6-14 strlen Character String/Memory Functions

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**.

6-15 strcoll Character String/Memory Functions

FUNCTION

strcoll compares two character strings based on the information specific to the locale.

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

- This compiler does not support operations specific to a cultural sphere. The operations are the same as that of **strcmp**.

6-16 strxfrm Character String/Memory Functions

FUNCTION

strxfrm converts a character string based on the information specific to the locale.

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

- This compiler does not support operations specific to a cultural sphere. The operations are the same as those of the following functions.

  strncpy (**s1**, **s2**, **c**) ;
  return (**strlen** (**s2**)) ;

7-1 acos Mathematical Functions

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 p).
- When **X** is non-numeric, **NaN** is returned.
- In the case of the definition area error of $x < -1$, $1 < x$, **NaN** is returned and **EDOM** is set.

7-2 asin Mathematical Functions

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 $x \leq -1$, $1 \leq x$, 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 an underflow occurs as a result of conversion, a non-normalized number is returned.

7-3 atan Mathematical Functions

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**

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 an underflow occurs as a result of conversion, a non-normalized number is returned.

7-4 atan2 Mathematical Functions

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** Non-normalized number ... When underflow occurs

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 an underflow occurs as a result of the operation, a non-normalized number is returned.

7-5 cos Mathematical Functions

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, x = ±∞ ... **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.

7-6 sin Mathematical Functions

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, x = ±∞ ... **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 an underflow occurs as a result of the 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.

7-7 tan Mathematical Functions

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 an underflow occurs as a result of the 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.

7-8 cosh Mathematical Functions

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** When overflow occurs, x = NaN, x = ±∞ ... **HUGE_VAL** (with positive sign) x = NaN ... **NaN**

EXPLANATION

- Calculates **cosh** of **x**.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is infinite, a positive infinite value is returned.
- If an overflow occurs as a result of the operation, **HUGE_VAL** with a positive sign is returned, and **ERANGE** is set to **errno**.

7-9 sinh Mathematical Functions

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 overflowed value) When underflow occurs ... ±0

EXPLANATION

- Calculates **sinh** of **x**.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is ±∞, ±∞ is returned.
- If an overflow occurs as a result of the operation, **HUGE_VAL** with the sign of the overflowed value is returned, and **ERANGE** is set to **errno**.
- If an underflow occurs as a result of the operation, ±0 is returned.

7-10 tanh Mathematical Functions

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 an underflow occurs as a result of the operation, ±0 is returned.

7-11 exp Mathematical

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 overflow occurs ... **HUGE_VQAL** (with positive sign) When underflow occurs ... Non-normalized number When annihilation of valid digits occurs due to underflow ... +0

EXPLANATION

- Calculates the exponent function of **x**.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is ±∞, ±∞ is returned.
- If an underflow occurs as a result of the operation, a non-normalized number is returned.
- If annihilation of valid digits due to underflow occurs as a result of the operation, +0 is returned.
- If an overflow occurs as a result of the operation, **HUGE_VAL** with a positive sign is returned and **ERANGE** is set to **errno**.

7-12 frexp Mathematical Functions

FUNCTION

frexp finds the 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

- Divides a floating-point number **x** into 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.

7-13 ldexp Mathematical Functions

FUNCTION

ldexp finds x*2^exp.

HEADER

math.h

FUNCTION PROTOTYPE

```
double ldexp (double x, int exp);
```

Function	Arguments	Return Value
exp	**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 overflowed 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 an overflow occurs as a result of the operation, **HUGE_VAL** with the overflowed value is returned and **ERANGE** is set to **errno**.
- If an underflow occurs as a result of the operation, a non-normalized number is returned.
- If annihilation of valid digits due to underflow occurs as a result of the operation, ±0 is returned.

7-14 log Mathematical Functions

FUNCTION

log finds the 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 ... **HUGE_VAL** (with negative sign) When x is non-numeric ... **NaN** When x is infinite ... +∞

EXPLANATION

- Finds the natural logarithm of **x**.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is +∞, +∞ is returned.
- In the case of an area error of **x** < 0, **HUGE_VAL** with a negative sign is returned, **EDOM** is set to **errno**.
- If **x** = 0, **HUGE_VAL** with a negative sign is returned, and **ERANGE** is set to **errno**.

7-15 log10 Mathematical Functions

FUNCTION

log10 finds the 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 ... **HUGE_VAL** (with negative sign) When x is non-numeric ... **NaN** When x is infinite ... +∞

EXPLANATION

- Finds the logarithm with 10 of **x** as the base.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is +∞, +∞ is returned.
- In the case of an area error of **x** < 0, **HUGE_VAL** with a negative sign is returned, **EDOM** is set to **errno**.
- If **x** = 0, **HUGE_VAL** with a negative sign is returned, and **ERANGE** is set to **errno**.

7-16 modf Mathematical Functions

FUNCTION

modf finds the fraction part and integer part.

HEADER

math.h

FUNCTION PROTOTYPE

```
double modif (double x, double *iptr);
```

Function	Arguments	Return Value
modif	**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** into a fraction part and integer part
- Returns the fraction part with the same sign as that of **x**, and stores the integer part in the location indicated by the pointer **iptr**.
- If **x** is non-numeric, **NaN** is returned and stored in the location indicated by the pointer **iptr**.
- If **x** is infinite, **NaN** is returned and stored in the location indicated by the pointer **iptr**, and **EDOM** is set to **errno**.
- If **x** = ±0, ±0 is stored in the location indicated by the pointer **iptr**.

7-17 pow Mathematical Functions

FUNCTION

pow finds the 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 underflow occurs ... Non-normalized number When overflow occurs ... **HUGE_VAL** (with the sign of overflowed value) When annihilation of valid digits occurs due to underflow ... ±0

EXPLANATION

- Calculates x^y.
- If an overflow occurs as a result of the operation, **HUGE_VAL** with the sign of overflown value is returned, and **ERANGE** is set to **errno**.
- 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 an underflow occurs, a non-normalized number is returned.
- If annihilation of valid digits occurs due to underflow, ±0 is returned.

7-18 sqrt Mathematical Functions

FUNCTION

sqrt finds the square root.

HEADER

math.h

FUNCTION PROTOTYPE

```
double sqrt (double x);
```

Function	Arguments	Return Value
sqrt	**x** ... Numeric value to perform operation	When x ≥ 0 ... Square root of **x** When x = ±0 ... ±0 When x < 0 ... **NaN**

EXPLANATION

- Calculates the square root of **x**.
- In the case of an 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.

7-19 ceil Mathematical Function

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 x = ±∞ ... **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.

7-20 fabs Mathematical Functions

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.

7-21 floor Mathematical Functions

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 x = ±∞ ... **NaN** When x = –0 ... +0 When the maximum integer no more than x cannot be expressed

EXPLANATION

- Finds the maximum integer no more 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 maximum integer no more than **x** cannot be expressed, **x** is returned.

7-22 fmod Mathematical Functions

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 ±0, when x is ±∞ ... **NaN** When x ≠ ∞ and y = ±∞ ... **x**

EXPLANATION

- Calculates the remainder of **x/y** expressed with **x** – i*__y__. i is an integer.
- If **y** ≠ 0, the return value has the same sign as that of **x** and the absolute value is less than that of **y**.
- If **y** is ± 0 or x = ±∞, **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.

7-23 matherr Mathematical Functions

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 and runtime libraries 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**.

7-24 acosf Mathematical Functions

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 $x \leq -1$, $1 < x$, x = ... **NaN**

EXPLANATION

- Calculates **acos** (range between 0 and π) of **x**
- If **x** is non-numeric, **NaN** is returned.
- In the case of a definition area error of $x \leq -1$, $1 \leq x$, **NaN** is returned and **EDOM** is set to **errno**.

7-25 asinf Mathematical Functions

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 $x \leq -1$, $1 < x$, x = NaN ... **NaN** x = –0 ... –0 When underflow occurs ... Non-normalized number

EXPLANATION

- Calculates **asin** (range between $-\pi/2$ and $+\pi/2$) of **x**
- If **x** is non-numeric, **NaN** is returned.
- In the case of definition area error of $x \leq -1$, $1 \leq x$, **NaN** is returned and **EDOM** is set to **errno**.
- If **x** = –0, –0 is returned.
- If an underflow occurs as a result of the operation, a non-normalized number is returned.

7-26 atanf Mathematical Functions

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**

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 an underflow occurs as a result of the operation, a non-normalized number is returned.

7-27 atan2f Mathematical Functions

FUNCTION

atan2f finds **atan** of **y/x**.

HEADER

math.h

FUNCTION PROTOTYPE

```
float atan2f (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 ±∞ ... **NaN** When underflow occurs ... Non-normalized number

EXPLANATION

- Calculates **atan** (range between $-\pi$ and $+\pi$) of y/x. When both **x** and **y** are 0 or the value whose **y/x** cannot be expressed, or when both **x** and **y** are infinite, **NaN** is returned and **EDOM** is set to **errno**.
- When either **x** or **y** is non-numeric, **NaN** is returned.
- If an underflow occurs as a result of the operation, a non-normalized number is returned.

7-28 cosf Mathematical Functions

FUNCTION

cosf finds **cos**.

HEADER

math.h

FUNCTION PROTOTYPE

```
float cost (float x);
```

Function	Arguments	Return Value
cosf	**x** ... Numeric value to perform operation	Normal ... **cos** of **x** When x = NaN, x = ±∞ ... **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.

7-29 sinf Mathematical Functions

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**, x = ±∞ ... **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 an underflow occurs as a result of the 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.

7-30 tanf Mathematical Functions

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**, 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 an underflow occurs as a result of the 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.

7-31 coshf Mathematical Functions

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** When overflow occurs, x = ±∞ ... **HUGE_VAL** (with a positive sign) x = **NaN** ... **NaN**

EXPLANATION

- Calculates **cosh** of **x**.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is infinite, positive infinite value is returned.
- If an overflow occurs as a result of the operation, **HUGE_VAL** with a positive sign is returned and **ERANGE** is set to **errno**.

7-32 sinhf Mathematical Functions

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** When overflow occurs ... **HUGE_VAL** (with a sign of the overflowed value) x = NaN ... **NaN** When x = ±∞ ... ±∞ When underflow occurs ... ±0

EXPLANATION

- Calculates **sinh** of **x**.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is ±∞, ±∞ is returned.
- If an overflow occurs as a result of the operation, **HUGE_VAL** with the sign of overflowed value is returned and **ERANGE** is set to errno.
- If an underflow occurs as a result of the operation, ±0 is returned.

7-33 tanhf Mathematical Functions

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 an underflow occurs as a result of the operation, ±0 is returned.

7-34 expf Mathematical Functions

FUNCTION

expf finds the 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** When overflow occurs ... **HUGE_VAL** (with positive sign) x = NaN ... **NaN** When x = ±∞ ... ±∞ When underflow occurs ... Non-normalized number When annihilation of valid 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 an overflow occurs as a result of the operation, **HUGE_VAL** with a positive sign is returned and **ERANGE** is set to **errno**.
- If an underflow occurs as a result of the operation, a non-normalized number is returned.
- If annihilation of effective digits occurs due to underflow as a result of the operation, +0 is returned.

7-35 frexpf Mathematical Functions

FUNCTION

frexpf finds the 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** into 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.

7-36 ldexpf Mathematical Functions

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 overflowed value) When underflow occurs ... Non-normalized numberV 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 overflowed value is returned and **ERANGE** is set to **errno**.
- If an underflow occurs as a result of the operation, a non-normalized number is returned.
- If annihilation of valid digits due to underflow occurs as a result of the operation, ±0 is returned.

7-37 logf Mathematical Functions

FUNCTION

logf finds the 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 is non-numeric ... **NaN** When x is infinite ... +∞ When x ≤ 0 ... **HUGE_VAL** (with negative sign)

EXPLANATION

- Finds natural logarithm of **x**.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is +∞, +∞ is returned.
- In the case of an area error of x < 0, **HUGE_VAL** with a negative sign is returned, and **EDOM** is set to **errno**.
- If x = 0, **HUGE_VAL** with a negative sign is returned, and **ERANGE** is set to **errno**.

7-38 log10f Mathematical Functions

FUNCTION

log10f finds the 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 is non-numeric ... **NaN** When x = +∞ ... +∞ When x ≤ 0 ... **HUGE_VAL** (with negative sign)

EXPLANATION

- Finds the logarithm with 10 of **x** as the base.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is +∞, +∞ is returned.
- In the case of an area error of x < 0, **HUGE_VAL** with a negative sign is returned, and **EDOM** is set to errno.
- If **x** = 0, **HUGE_VAL** with a negative sign is returned, and **ERANGE** is set to **errno**.

7-39 modff Mathematical Functions

FUNCTION

modff finds the 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** into a fraction part and integer part.
- Returns the fraction part with the same sign as that of **x**, and stores the integer part in the location indicated by the pointer **iptr**.
- If **x** is non-numeric, **NaN** is returned and stored in the location indicated by the pointer **iptr**.
- If **x** is infinite, **NaN** is returned and stored in the location indicated by the pointer iptr, and **EDOM** is set to **errno**.
- If **x** = ±0, ±0 is returned and stored in the location indicated by the pointer **iptr**.

7-40 powf Mathematical Functions

FUNCTION

powf finds the 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 underflow occurs ... Non-normalized number When overflow occurs ... **HUGE_VAL** (with the sign of overflowed value) When annihilation of valid digits occurs due to underflow ... ±0

EXPLANATION

- Calculates **x^y**.
- If an overflow occurs as a result of the operation, **HUGE_VAL** with the sign of overflowed value is returned, and **ERANGE** is set to **errno**.
- 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 an underflow occurs, a non-normalized number is returned.
- If annihilation of valid digits occurs due to underflow, ±0 is returned.

7-41 sqrtf Mathematical Functions

FUNCTION

sqrtf finds the square root.

HEADER

math.h

FUNCTION PROTOTYPE

```
float sqrtf (float x);
```

Function	Arguments	Return Value
sqrtf	**x** ... Numeric value to perform operation	When **x** ≥ 0 ... Square root of **x** When **x** = ±0 ... ±0 When **x** < 0 ... **NaN**

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.

7-42 ceilf Mathematical Functions

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 x = ±∞ ... **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.

7-43 fabsf Mathematical Functions

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.

7-44 floorf Mathematical Functions

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 –0, +0 is returned.
- If **x** is infinite, **NaN** is returned and **EDOM** is set to **errno**.
- If the maximum integer no more than **x** cannot be expressed, **x** is returned.

7-45 fmodf Mathematical Functions

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** is non-numeric or **y** is non-numeric When **y** is ±0, when **x** is ±∞ ... **NaN** When **x** ≠ ∞ and **y** = ±∞ ... **x**

EXPLANATION

- Calculates the remainder of **x/y** expressed with **x** – i***y.** i is an integer.
- If **y** ≠ 0, the return value has the same sign as that of **x** and the absolute value is less than **y**.
- If **y** is ± 0 or x = ±∞, **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.

8-1 _ _ assertfail Diagnostic Functions

FUNCTION

_ _ **assertfail** supports the **assert** macro.

HEADER

math.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

The _ _ **assertfail** function receives information from the **assert** macro (refer to **10.2 Headers (13) assert.h**), calls the **printf** function, outputs information, and calls the **abort** function.

The **assert** macro adds diagnostic functions 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 the _ _**assertfail** function.

10.5 Batch Files for Update of Startup Routine and Library Functions

This compiler is provided with batch files for updating a part of the standard library functions and the startup routine. The batch files in the BAT directory are shown in Table 10-3 below.

Caution The file d4025.78k in the BAT directory is used during batch file activation for updating the library, not for development. When developing a system, it is necessary to have a device file (sold separately).

Table 10-3. Batch Files for Updating Library Functions

Batch File	Application
mkstup.bat	Updates the startup routine (cstart*.asm). When changing the startup routine, perform assembly using this batch file.
reprom.bat	Updates the firmware 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 SM78K4-supporting. When it is necessary to check the output of the **putchar** function using the SM78K4, 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 4000H. 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.

10.5.1 Using batch files

Use the batch files in the subdirectory BAT. Because these files are the batch files used to activate the assembler and librarian, an environment in which the assembler package RA78K4 Ver. 1.50 or later operates is necessary.

Before using the batch files, set the directory that contains the RA78K4 execution format file using the environment variable PATH.

Create a subdirectory (LIB) of the same level as BAT for the batch files and put the post-assembly files in this subdirectory.

When a C startup routine or library is installed in a subdirectory LIB that is the same level as BAT, these files are overwritten.

To use the batch files, move the current directory to the subdirectory BAT and execute each batch file. At this time, the following parameters are necessary.

Product type = chiptype (classification of target chip)
4026 ··· μPD784026, etc.

The following is an illustration of how to use each batch file.

The batch file for:

(1) Startup routine

- For PC-9800 series, IBM PC/AT™ and compatibles
  mkstup chiptype

Example `mkstup 4026`

- For HP9000 series 700™, SPARCstation™ Family
  /bin/sh mkstup.sh chiptype

Example `/bin/sh mkstup.sh 4026`

(2) Firmware ROM routine update

- For PC-9800 series, IBM PC/AT and compatibles
  reprom chiptype

Example `reprom 4026`

- For HP9000 series 700, SPARCstation Family
  /bin/sh reprom.sh chiptype

Example `/bin/sh reprom.sh 4026`

(3) getchar function update

- For PC-9800 series, IBM PC/AT and compatibles
  repgetc chiptype

Example `repgetc 4026`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repgetc.sh chiptype

Example `/bin/sh repgetc.sh 4026`

(4) putchar function update

- For PC-9800 series, IBM PC/AT and compatibles
  repputc chiptype

Example `repputc 4026`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repputc.sh chiptype

Example `/bin/sh repputc.sh 4026`

(5) putchar function (SM78K4-supporting) update

- For PC-9800 series, IBM PC/AT and compatibles
  repputcs chiptype

Example `repputcs 4026`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repputcs.sh chiptype

Example `/bin/sh repputcs.sh 4026`

(6) setjmp/longjmp function update (with restore/save processing)

- For PC-9800 series, IBM PC/AT and compatibles
  repselo chiptype

Example `repselo 4026`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repselo.sh chiptype

Example `/bin/sh repselo.sh 4026`

(7) setjmp/longjmp function update (without restore/save processing)

- For PC-9800 series, IBM PC/AT and compatibles
  repselon chiptype

Example `repselon 4026`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repselon.sh chiptype

Example `/bin/sh repselon.sh 4026`

(8) Interrupt vector table update

- For PC-9800 series, IBM PC/AT and compatibles
  repvect chiptype

Example `repvect 4026`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repvect.sh chiptype

Example `/bin/sh repvect.sh 4026`

CHAPTER 11 EXTENDED FUNCTIONS

This chapter describes the extended functions unique to this C compiler and not specified in the **ANSI** (American National Standards Institute) **Standard** for C.

The extended functions of this C compiler are used to generate codes for effective utilization of the target devices in the 78K/IV Series. Not all of these extended functions are always effective. Therefore, it is recommended to use only the effective ones according to the purpose of use. 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 C compiler 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” indicates the 78K/IV Series real-time OS.

11.1 Macro Names

This C compiler has two types of macro names: those indicating the series name for target devices and those indicating device name (processor type). These macro names are specified according to the option for compilation to output object code for a specific target device or according to the processor type in the C source. In the example below, _ _**K4**_ _ and _ _**4026**_ are specified.

For details of these macro names, see **9.8 Compiler-Defined Macro Names.**

[Example]

Option for compilation

```
>CC78K4  -C4026  prime.c …
```

Specification of device type:

```
#pragma pc (4026)
```

11.2 Keywords

The following tokens are added to this C compiler as keywords to realize the extended functions. Similarly to ANSI-C keywords, these tokens cannot be used as labels or as variable names. All the keywords must be described in lowercase letters. A keyword containing an uppercase letter is not interpreted as a keyword by the C compiler.

This following shows the list of keywords added to this compiler. 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 **2.1 Keywords**).

Table 11-1. List of Added Keywords

Keyword		Use
_ _callt	**callt**	**callt/_ _callt** functions
_ _callf	**callf**	**callf/_ _callf** functions
_ _sreg	**sreg**	**sreg/_ _sreg** variables
_ _sreg1		**_ _sreg1** variables
	noauto	**noauto** functions
_ _leaf	**norec**	**norec/_ _leaf** functions
_ _boolean	**boolean**	**boolean** type/_ **_boolean** type
	bit	**bit** type variables
_ _boolean1		**_ _boolean1** type variables
_ _interrupt		Hardware interrupt
_ _interrupt_brk		Software interrupt
_ _asm		ASM statements
_ _rtos_interrupt		Handler to allocate for RTOS
_ _pascal		Pascal function
_ _flash		Firmware ROM function
_ _directmap		Absolute address allocation specification

(1) Functions

The keywords **callt, _ _callt, callf _ _callf, noauto, norec, _ _leaf, _ _interrupt, _ _interrupt_brk, _ _rtos_interrupt**, and **_ _flash** 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)

```
_ _callt  int  func  (int);
```

Attribute qualifier specifications are limited to those listed below. (The **noauto** and **norec/_ _leaf** qualifiers cannot be specified at the same time.) **callt** and _ _**callt**, **callf** and _ _**callf**, **norec** and _ _**leaf** are regarded as the same specifications. However, qualifiers that include '_ _' are enabled even when the **-ZA** option is specified.

- callt
- callf
- noauto
- norec
- callt noauto
- callt norec
- noauto callt
- norec callt
- callf noauto
- callf norec
- noauto callf
- norec callf
- _ _interrupt
- _ _interrupt_brk
- _ _rtos_interrupt
- _ _pascal
- _ _pascal noauto
- _ _pascal callt
- _ _pascal callf
- noauto_ _pascal
- callt_ _pascal
- callf_ _pascal
- callt noauto_ _pascal
- callf noauto_ _pascal
- _ _flash

(2) Variables

- The keyword **sreg, _ _sreg, or _ _sreg1** is specified in a similar manner to the **register** storage class specifier of C. (For details, see **11.5 (3) How to use the saddr area.**)
- The keyword **bit**, **boolean, _ _boolean,** or _ _**boolean1** is specified in a similar manner to the **char** or **int** type specifier of C.
  However, these types can be specified only for the variables defined outside a function (external variables).
- The same regulations apply to the _ _**directmap** specification as to the type qualifiers in C language (refer to **11.5 (42) Absolute address allocation specification** for details).

11.3 Memory

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

(1) Memory model

A maximum of 1 MB of program memory space and a maximum of 16 MB of data memory space are available (for the memory map, refer to the user's manual of each target device).
This compiler has the three types of memory models: small, medium, and large. Objects are changed and output by specifying each memory model option. For details of each model, refer to **Table 11-2**.

Table 11-2. Memory Model

Memory Model (Option)	Explanation
Small model (**-MS**)	A model with a combined code/data block capacity of 64 KB.
Medium model (**-MM**)	A model with a capacity of up to 1 MB for the code block and 64 KB for the data block
Large model (**-ML**)	A model with a combined code/data block capacity of 16 MB, including up to 1 MB for the code block and 16 MB for the data block.

(2) Register bank

- The register bank is set to 'RB0' at startup (set in the startup routine of this compiler). 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) Location function

- With the large model or medium model, the location function (**-CS** option) allows changing the location of the internal RAM (including **saddr** area and **sfr** area) between 64 KB (LOCATION 00H) and 1024 KB (LOCATION 0FH) (with the small model, the location of the internal RAM is fixed to 64 KB). For the **-CS** option, refer to the **CC78K4 C Compiler Operation User's Manual (U15557E)**.

(4) Memory space

This C compiler uses memory space as shown in Table 11-3 below.

Table 11-3. Utilization of Memory Space

Address		Use	Size (Bytes)
00	40 to 7FH	CALLT table	64
0800 to 0FFFH		CALLF entry	2048
(F)FD	20 to DFH	sreg variables, **boolean** type variables	192
(F)FD	20 to FFH	Arguments of **norec** functions[Note 1]	8
Consecutive 32-byte area in the interval above		Automatic variables of **norec** functions[Note 1]	8
		Register variables[Note 1]	16
(F)FE	00 to 7FH	**sreg1** variables, **boolean1** type variables	128
(F)FE	80 to EFH	RB7 to RB1[Note 2] (work registers)	112
	F0 to FFH	RB0 (work registers)	16
(F)FF	00 to FFH	**sfr** variables	256

Notes 1. The restore to this area is not processed within the interrupt function when the **-qr** option is not specified (default). This reduces the preprocessing/postprocessing of interrupt functions and allows users to use the areas of **Note 1** as **sreg** variable or **boolean** type variable areas when using a real-time OS, etc. For the save/restore processing code output, refer to **11.5 (10) Interrupt function**. This area, as shown in **APPENDIX A LIST OF LABELS FOR saddr AREA**, defines labels and secures areas in a library.

Standard library functions **setjmp**, **longjmp** refer to a part of this area **_@KREG00**.

2. Used when a register bank is specified.

11.4 #pragma directives

The **#pragma** directives are preprocessing directives supported by ANSI. A **#pragma** directive, depending on the character string to follow **#pragma**, instructs the compiler to translate using the method determined by the compiler. If the compiler does not support **#pragma** directives, the **#pragma** directive is ignored and compilation is continued. If keywords are added by a directive, an error is output if the C source includes the keywords. In order to avoid this, the keywords in the C source should either be deleted or sorted by the **#ifdef** directive.

This C compiler 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 11-4. List of #pragma Directives

#pragma Directive	Applications
#pragma sfr	Describes SFR name in C → **11.5 (4) How to use the sfr area**
#pragma asm	Inserts ASM statement in C source → **11.5 (9) ASM statements**
#pragma vect #pragma interrupt	Describes interrupt processing in C → **11.5 (10) Interrupt functions**
#pragma di #pragma ei	Describes DI/EI instructions in C → **11.5 (12) Interrupt functions**
#pragma halt #pragma stop #pragma nop #pragma brk	Describes CPU control instructions in C → **11.5 (13) CPU control instruction**
#pragma access	Uses absolute address access functions → **11.5 (17) Absolute address access function**
#pragma section	Changes compiler output section name and specifies section location → **11.5 (19) Changing compiler output section name**
#pragma name	Changes module name → **11.5 (21) Module name changing function**
#pragma rot	Uses rotate function → **11.5 (22) Rotate function**
#pragma mul	Uses multiplication function → **11.5 (23) Multiplication function**
#pragma div	Uses division function → **11.5 (24) Division function**
#pragma opc	Uses data insertion function → **11.5 (25) Data insertion function**
#pragma rtos_interrupt	Uses interrupt handler for real-time OS (RX78K/IV) → **11.5 (26) Interrupt handler for real-time OS (RTOS)**
#pragma rtos_task	Uses task function for real-time OS (RX78K/IV) → **11.5 (28) Task function for real-time OS (RTOS)**
#pragma ext_table	Specifies the first address of the flash area branch table → **11.5 (34) Flash area branch table**
#pragma ext_func	Calls a function to the flash area from the boot area → **11.5 (35) Function call function from the boot area to the flash area**.
#pragma inline	Expands the standard library functions **memcpy** and **memset** inline → **11.5 (38) Memory manipulation function**
#pragma addnaccess	Uses 3-byte address reference/generation function → **11.5 (41) Three-byte address reference/generation function**

11.5 How to Use Extended Functions

This section describes the extended functions in the following format.

FUNCTION:
Outlines the 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:
Gives 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 this C compiler.

(1) callt functions

callt Functions — callt/_ _callt

FUNCTION

- The **callt** instruction stores the address of a function to be called in an area [40H to 7FH] 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 */
          :
}
```

callt Functions — callt/_ _callt

RESTRICTIONS

- 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 40F to 70F.
- When the **callt** table is used exceeding the number of **callt** attribute functions permitted, a compilation error will occur.
- The **callt** table is used by specifying the **-QL** option. For that reason, the number of callt attributes permitted per load module and the total in the linking modules is as shown in Table 11-5.

Table 11-5. Number of callt Attribute Functions That Can Be Used When –QL Option Is Specified

Memory Model	Number of Functions That Can Be Used			
	-QL1	-QL2	-QL3	-QL4
Small model	32	32	15	10
Medium model	32	25	8	3
Large model	32	23	6	1

- Cases in which the **-QL** option is not used and the defaults are as shown below.

Table 11-6. Restriction on callt Function Usage

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 main ()                        _ _callt int tsub ()
{                                   {
        int ret_val;                      int val;
       ret_val = tsub();                  return val;
}                                   }
```

(Output object of assembler)

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

ca2 module
        PUBLIC    _tsub              ;Declaration
        PUBLIC    ?tsub              ;
@@CALT  CSEG        CALLT0           ;Allocation to segment
?tsub:  DW        _tsub
@@CODE CSEG
_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 this C compiler

- Modification is not required if the keyword **callt**/_ _ **callt** is not used.
- When changing functions to **callt** functions, use the method above.

From this C compiler to another C compiler

- **#define** must be used. For details, see **11.6 Modifications of C Source.**

(2) Register variables

Register Variables register

FUNCTION

- Allocates the declared variables (including arguments of function) to the register (RP3, VP) and **saddr2** area **(_@KREG00** to **_@KREG15**). Saves and restores registers or **saddr2** area during the preprocessing/ postprocessing of the module that declared a register.
- When the **-ZO** option is specified, register variables are allocated in the order of declaration. When the **-ZO** option is not specified (default), on the other hand, the allocation is performed based on the number of references. Therefore, it is undefined to which register or **saddr2** area the register variable is allocated. For details of the allocation of register variables, refer to **11.7 Function Call Interface**.
- Register variables are allocated to different areas depending on the compilation condition as shown below (for each option, refer to the **CC78K4 C Compiler Operation User's Manual (U15557E)**).

1. Register variables are allocated to **saddr2** area only when the **-QR** option is specified.
2 When the **-QF** option is specified and the **-ZO** option is not specified, register variables are also allocated also to register **UP**.
3. When neither the **-ZO** option nor the **-QF** option is specified, all the register arguments and register variables are allocated to registers and **saddr2** area. When there is no argument or automatic variable allocated to the stack area (that is, a stack frame is not generated), register variables are also allocated to register **UP** (when the **-ML** option is specified and the **-QR** option is not specified, however, register variables are allocated only if the total size allocated to the register is 6 bytes or less assuming the pointer is 3 bytes).

Register Variables register

These are summarized in Table 11-7.

Table 11-7. Registers to Allocate Register Variables

Without –ZO

Option Specification	Registers to Allocate
Without **-QR**	RP3, VP
With **–QR**	RP3, VP, saddr2 area (_@KREG00 to _@KREG15)
With **-QF** *1	RP3, VP, UP
Without **-QF** and a stack frame not generated *2	RP3, VP, UP
Above *1 or *2 and with **-QR**	RP3, VP, UP, saddr2 area (_@KREG00 to _@KREG15)

With –ZO

Option Specification	Registers to Allocate
Without **-QR**	RP3, VP
With **-QR**	RP3, VP, saddr2 area (_@KREG00 to _@KREG15)
With **-QF** *1	RP3, VP
Without **-QF** and a stack frame not generated *2	RP3, VP
Above *1 or *2 and with **-QR**	RP3, VP, saddr2 area (_@KREG00 to _@KREG15)

EFFECT

- Instructions to the variables allocated to the register or **saddr2** area are generally shorter in code length than those to memory. This helps shorten object code and also improves program execution speed.

Register Variables register

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.
- With the medium model, function pointers are allocated to **saddr2** area for register variables. Function pointers cannot be allocated to registers.
- A **char** type register variable uses only half the space required for the register variable of any other type. A **long/float/double/long double** type variable uses twice the space.
- The function pointer type of the medium model and the pointer of the large model use one and a half the amount of space.
- All the types have byte boundaries.
- If the register variables exceed the 'usable number' shown in **Table 11-8**, they are handled the same as automatic variables without a register storage class specifier and allocated to ordinary memory space.
- Up to 20 bytes or 22 bytes can be allocated as register variables (6 bytes when 16 bytes of **saddr2** area and 4 bytes of registers or **UP** are used).

Table 11-8. Restrictions on Register Variables Usage

Data Type	Usable Number (Per Function)
int/short	10 variables max.
Function pointer of medium model	5 variables max.
Pointer of large model	6 variables max.
long/float/double/long double	5 variables max.

Register Variables register

EXAMPLE 1

1. Example of register variable allocation to register
   (With the large model, and when the optimization option is the default)

(C source 1)

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

(Output object of compiler)

```
   @@CODE CSEG
   _main:
           push        uup          ;Saves register contents at the beginning of the function.
           Push        rp3          ;
           subw        rp3, rp3     ;Assigns 0 to i
           movw        up, #01H     ;Assigns 1 to j
           addw        rp3, up      ;Assigns i to the result of i + j
           pop rp3                  ;Restores register contents at the end of the function.
           Pop uup                  ;
           ret
```

Register Variables register

EXAMPLE 2

2. Example of register variable allocation to register and **saddr2** area
   (With the large model, and when the optimization option **-QR** is specified)

(C source 2)

```
   void main () {
         register unsigned int a, b, c, d;
         d = a - b;
         d = b - c;
}
```

(Output object of compiler)

```
    EXTRN   SADDR2(_@KREG00)      ; Performs reference declaration of saddr2 area to be used.
    PUBLIC  _main
@@CODE      CSEG
_main;
    push    uup                   ; Saves register contents at the beginning of the function.
    push    rp3                   ;
    push    vvp                   ;
    movw    ax, _@KREG00          ; Saves contents of saddr2 are at the beginning of the function.
    push    ax
    movw    ax, rp3               ;
    subw    ax, up                ; a – b
    movw    vp, ax                ; Assigns the result of a – b to d
    movw    ax, up
    subw    ax, _@KREG00          ; b – c
    movw    vp, ax                ; Assigns the result of b – c to d
    pop     ax
    movw    _@KREG00, ax          ; Restores contents of saddr2 area at the end of the function.
    pop     vvp
    pop     rp3
    pop     uup                   ; Restores register contents at the end of the function.
    ret
```

Register Variables register

EXPLANATION

- To use register variables, you only need to declare them with the **register** storage class specifier.
- Label _**@KREG00**, etc. includes the modules declared with **PUBLIC** in the library attached to this C compiler.

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the other C compiler supports **register** declarations.
- When changing to **register** variables, add the register declarations for the variables to the program.

From this C compiler to another C compiler

- Modification is not required if the other compiler supports **register** declarations.
- How many variable registers can be used and to which area they will be allocated depends on the implementation of the other C compiler.

(3) How to use the saddr area

Usage of saddr Area — sreg/_ _sreg

(1) Usage with sreg declaration

FUNCTION

- The external variables and in-function **static** variables (**sreg** variables) declared with the keyword **sreg** or _ _**sreg** are automatically allocated to **saddr2** [XFD20H to XFDFFH] area with relocatability (X: 0 or F by specifying location). When those variables exceed the area shown above, a compilation error occurs.
- 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 a **boolean** type variable automatically.
- **sreg** variables declared without an initial value take 0 as the initial value.
- The area of **sreg** variables declared in the assembler source that can be referenced is the **saddr2** area [XFD20H to XFDFFH]. When the **-QR** option is specified, however, the compiler may use up to 32 MB of **saddr2** area, so care must be taken (refer to **Table 11-3 Utilization of Memory Space**).

EFFECT

- Instructions to the **saddr2** 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 that defines the variables. Only a 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 that 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
```

Usage of saddr Area sreg/_ _sreg

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.

Arguments of functions and automatic variables cannot be specified to this area.

char type uses half the space of other types and **long/float/double/long double** types use twice the space. Function pointers of the medium model and the large model use one and a half the amount of space as other types.

All the types have byte boundaries.

When **-ZA** is specified, only _ _**sreg** is enabled and **sreg** is disabled.

The following shows the maximum number of **sreg** variables that can be used per load module.

Table 11-9. Restrictions on sreg Variable Usage

Data Type	Usable Number of sreg Variables (Per Load Module)
int/short	Max. 112 (96 when -QR is specified)[Note]
Function pointer of medium model	Max. 74 (64 when -QR is specified)[Note]
Pointer of large model	Max. 74 (64 when -QR is specified)[Note]

Note When the **-QR** option is not specified, the reserved area for the argument of the **norec** function/automatic variables and register variables (32 bytes of **saddr2** area) can be used as **sreg** variable area. When **bit** and **boolean** type variables are used, the usable number is decreased.

EXAMPLE

The following shows an example when the large model is used.

(C source)

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

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

Usage of saddr Area sreg/_ _sreg

(Assembler source)

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

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

@@DATS      DSEG    SADDR2             ;Allocation to segment
            ORG     0FFD20H            ;
_hsmm0:             DS    (2)          ;
_hsmm1:             DS    (2)          ;
_hsptr:             DS    (3)          ;
```

(Output object of compiler)

```
        EXTRN     SADDR2(_hsmm1)
        EXTRN     SADDR2(_hsmm0)
        PUBLIC   _main

@@CODE          CSEG
_main:
       subw      _hsmm0, _hsmm1
       ret
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the other compiler does not use the keyword **sreg**/_ _**sreg**.
  When changing to **sreg** variable, use the method above.

From this C compiler to another C compiler

- Modifications are made by **#define**. For details, refer to **11.6 Modifications of C Source**. By this modification, **sreg** variables are handled as ordinary variables.

Usage of saddr Area -RD

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

FUNCTION

- External variables/external **static** variables (except **const** type) are automatically allocated to the **saddr2** area regardless of whether an **sreg** declaration is made or not.
- Depending on the value of n, the external variables and external **static** variables to allocate can be specified as follows.

Table 11-10. Variables Allocated to saddr2 Area by -RD Option

Value of n	Variables Allocated to saddr2 Area
If 1	Variables of **char** and **unsigned char** types
If 2	Variables if n is 1 and variables of **short, unsigned short, int, unsigned int, enum**, small model pointer, and medium model data pointer type
If 4	Variables if n is 2 and variables of **long, unsigned long, float, double, long double**, medium model pointer, and large model pointer type
If omitted	All variables (including structures, unions, and arrays in this case only)

- Variables declared with the keyword **sreg** are allocated to the **saddr2** area, regardless of the above specification.
- The above rule also applies to variables referenced by an **extern** declaration, and processing is performed as if these variables were allocated to the **saddr2** area.
- The variables allocated to the **saddr2** 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 (1).

METHOD OF SPECIFICATION

Specify the **-RD [n]** (**n**: 1, 2, or 4) option.

RESTRICTIONS

- With the **-RD [n]** option, modules specifying different **n** values cannot be linked to each other.

Usage of saddr Area -RS

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

FUNCTION

- Automatically allocates internal **static** variables (except **const** type) to **saddr2** area regardless of an **sreg** declaration.
- Depending on the value of n, the internal static variables to allocate can be specified as follows.

Table 11-11. Variables Allocated to saddr2 Area by -RS Option

Value of n	Variables Allocated to saddr2 Area
If 1	Variables of **char** and **unsigned char** types
If 2	Variables if n is 1 and variables of **short**, **unsigned short**, **int**, **unsigned it**, **enum**, small model pointer, and medium model data pointer type
If 4	Variables if n is 2 and variables of **long**, **unsigned long**, **float**, **double**, **long double**, medium model function pointer, and large model pointer type
If omitted	All variables (including structures, unions, and arrays in this case only)

- Variables declared with the keyword **sreg** are allocated to the **saddr2** area regardless of the above specification.
- The variables allocated to the **saddr2** 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 (1).

METHOD OF SPECIFICATION

Specify the **-RS [n]** (**n**: 1, 2, or 4) option.

Remark With the **-RS [n]** option, modules specifying different **n** values can also be linked to each other.

Usage of saddr Area _ _sreg1

(4) Usage with _ _sreg1 declaration

FUNCTION

- Variables declared with the keyword _ _**sreg1** (called **sreg1** variables) are automatically allocated to **saddr1** [XFE00H to XFE7FH] area (x: 0 or F by specifying location) with relocatability. When the **sreg1** variable exceeds the area shown above, a compilation error occurs.
- **saddr1** area [XFE00H to XFEFFH] can be used as **sreg1** variables by changing the location of segments in the assembler source or at the time of linking. However, care must be taken because the compiler uses the area [XFE80H to XFEFFH] as a general-purpose register area.
- The **sreg1** variables are handled in the same manner as ordinary variables in the C source.
- Each bit of **sreg1** variables of **char/short/int/long** type automatically becomes a _ _**boolean1** type variable.
- **sreg1** variables declared without an initial value take 0 as the initial value.

EFFECT

- Instructions to the **saddr1** area are generally shorter in code length than those to memory. This helps shorten object code and also improves program execution speed.

USAGE

- Declare a variable with the keyword _ _**sreg1** inside the module in which the variable is to be defined.

```
_ _sreg1 type-name variable-name
```

- Declare the following variables inside the module in which the **sreg1** variable is referenced.

```
extern _ _sreg1 type-name variable-name
```

Usage of saddr Area _ _sreg1

RESTRICTIONS

- When _ _**sreg1** type is specified for a **const** type or function, a warning message is output and the _ _**sreg1** declaration is ignored.
- Arguments of functions and automatic variables cannot be specified to this area.
- **char** type uses half the space of other types, and **long/float/double/long double** types use twice the space.
- All the types have byte boundaries.
  Medium model function pointers and large model pointers use one and a half the space of other types.

The following shows the maximum number of **sreg** variables that can be used per load module.

Table 11-12. Restrictions on sreg1 Variable Usage

Data Type	Usable Number of sreg Variables (Per Load Module)
int/short	Max. 64[Note]
Medium model function pointer	Max. 42[Note]
Large model pointer	Max. 42[Note]

Note **saddr1** area [XFE00H to XFE7FH] is used. When _ _**boolean1** type variables are used, the usable number is decreased.

EXAMPLE

The following shows an example when the large model is used.

(C source)

```
extern  _ _sreg1 int  s1;
extern  _ _sreg1 int  s2;
extern  _ _sreg1 int  *spr;

void main( ) {
      s1 -= s2;
}
```

Usage of saddr Area _ _sreg1

(Assembler source)

The following example shows a definition code for a **sreg1** variable that the user creates. If an **extern** declaration is not made in the C source, the C compiler outputs the codes in the same way as those of assembler source. In this case, the **ORG** quasi directive will not be output.

```
                PUBLIC _s1              ;Declaration
                PUBLIC _s2              ;
                PUBLIC _sptr            ;

@@DATS1 DSEG SADDR                      ;Allocation to segment
        ORG    0FFE00H                  ;
_s1:    DS     (2)                      ;
_s2:    DS     (2)                      ;
_sptr:  DS     (3)                      ;
```

(Output object of compiler)

```
        EXTRN   _s2
        EXTRN   _s1
        PUBLIC  _main

        @@CODE  CSEG
        _main:
                subw   _s1,_s2
                ret
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the keyword _ _**sreg1** is not used in the program.
- When changing to **sreg1** variables, use the method above.

From this C compiler to another C compiler

- **#define** must be used. For details, see **11.6 Modifications of C Source**. By this modification, **sreg1** variables will be handled as ordinary variables.

(4) How to use the sfr area

Usage of 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 78K/IV Series 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 values (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 illegal data to an **sfr** variable will result in a compilation error.
- The **sfr** names that can be used are those allocated to an area consisting of addresses FF00H to FFFFH with the small model, or XFF00H to XFFFFH with the medium large model. (x: 0 or F by specifying location)

EFFECT

- Manipulations on the **sfr** area can be described at 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** preprocessing directive, as follows. (The keyword **sfr** can be described in uppercase or lowercase letters.).

```
#pragma sfr
```

- The **#pragma sfr** directive must be described at the beginning of the C source line. If **#pragma PC** (processor type) is specified, however, describe **#pragma sfr** after that.
  The following statement and directives may precede the **#pragma sfr** directive.
  . Comment statement
  . Preprocessing directives that do not define or 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.

Usage of sfr Area sfr

RESTRICTIONS

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

EXAMPLE

(C source)

```
#ifdef _ _K4_ _
      #pragma sfr
#endif

void main ()
{
      CMK00 = 1;
      PM0 = 0x11;
      P0 = 10;
      :
}
```

(Output object of compiler)

The C compiler outputs no declaration-related code but outputs the following code inside the function.

```
@@CODE CSEG
_main:
      set1   CIC00.6
      mov    PM0, #011H      ;17
      sub    P0, #0AH        ;10
      ret
```

Usage of sfr Area sfr

EXPLANATION

- In the above example, use of **sfr** variables is declared with the **#pragma sfr** directive. By this declaration, special function registers such as P0 (port 0) and CIC00 (one of the interrupt control registers[Note]) can be used.

  Note Bit 6 of the CIC00 register has the SFR bit name CMK00. For **sfr**, refer to the user's manual of the target device used.

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if those portions of the C source program do not depend on the device or compiler.

From this C compiler to another C compiler

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

```
#ifdef _ _K4_ _
    #pragma sfr
#else
/* declaration of variables */
unsigned char P0;
#endif

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

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

(5) noauto function

noauto Function noauto

FUNCTION

- The **noauto** function sets restrictions for automatic variables not to output the codes of preprocessing/ postprocessing (generation of stack frame).
- All the arguments are allocated to registers. If there is an argument that cannot be allocated to registers, a compilation error occurs.

(a) When -ZO option is specified

- Arguments are passed via registers.
- The locations where arguments are passed to the function call side and the function definition side become the locations where arguments are allocated.
- The save and restore of the register to which arguments are allocated are performed before/after the function call.
- Automatic variables cannot be used.
- Arguments are allocated in the same order as ordinary functions.
- Table 11-13 shows the registers to which the arguments of the noauto function are passed/allocated.

Table 11-13. Registers Used for noauto Function Arguments (With -ZO)

Data Type	First Argument	Second Argument	Third Argument or Later
char	R6	R7	R8, R9, R10, R11
int, short	RP3	VP	UP (only when **–MS –QF** is specified)
long/float/double/ long double	VP (higher 16 bits) RP3 (lower 16 bits)		
Small model pointer	VP	UP (only when **–QF** is specified)	RP3
Large model pointer	VVP		

noauto Function noauto

(b) When -ZO option is not specified

- Arguments are passed on the function call side in the same manner as ordinary functions (refer to **11.7.2 Ordinary function call interface**).
- The arguments passed via a register or stack are copied to the register shown in **Table 11-14** on the function definition side (copying register is necessary even when an argument is passed via a register because the registers of the function call side and the function definition side are different).
- The save and restore of registers to which arguments are allocated are performed on the function definition side.

Table 11-14. Registers Used for noauto Function Arguments (Without -ZO)

Data Type	First Argument	Second Argument	Third Argument or Later
char (with 4-byte argument) [Note] char (without 4-byte argument) [Note]	R10 R6	R11 R7	R6, R7, R8, R9, R10, R11, R8, R9
int, short, enum (with 4-byte argument)[Note] (without 4-byte argument)[Note]	 UP RP3	 RP3 UP	 VP VP
long/float/double/long double	VP (higher 16 bits) RP3 (lower 16 bits)		
Small model pointer Medium model data pointer	UP	VP	RP3
Large model pointer	UUP	VVP	

Note 4-byte arguments are arguments of **long, float, double, long double** type

Remarks 1. The medium model function pointer cannot be used as an argument to be allocated to a register.

2. The order of the register allocation in this function is the same as the order when the **-QF** option is specified in ordinary functions.

noauto Function noauto

- Automatic variables can be used only when all the automatic variables can be allocated to the registers remaining after the argument allocation and to the **saddr2** area (**_@KREGXX**) for register variables. However, automatic variables are allocated to the **saddr2** area for register variables only when the **-QR** option is specified during compilation. If the **-QRO** option is specified during compilation, a warning message is output and automatic variables are not allocated to **saddr2** area.
- Automatic variables are allocated in the same order as arguments are allocated. The automatic variables allocated to **saddr2** area (**_@KREGXX**) are allocated in the order of declaration (if they are not allocated, a compilation error occurs).
- The save and restore of **_@KREGXX**, the register to which automatic variables are allocated, are performed on the function definition side.

EFFECT

- The object code can be shortened and execution speed can be improved.

USAGE

Declare a function with the **noauto** attribute in the function declaration, as follows.

```
noauto type-name function-name
```

RESTRICTIONS

- When the **-ZO** option is specified, automatic variables cannot be used inside the **noauto** function, and neither can the register variables.
- When the **-ZA** option is specified, the **noauto** function is disabled.
- The arguments and automatic variables of the **noauto** function (only when the **-ZO** option is specified) have restrictions on their types and numbers. The following shows the types of arguments that can be used inside a **noauto** function.

- Pointer
- `char / signed char/ unsigned char`
- `int / signed int / unsigned int`
- `short / signed short / unsigned short`
- `enum`
- `long / signed long / unsigned long`
- `float / double / long double`

noauto Function noauto

Table 11-15. Restrictions on noauto Function Arguments (With -ZO)

Data Type	Restriction
Type other than pointer	Max. 4 bytes (Max. 6 bytes)[Note]
Small model pointer	Max. 4 bytes (Max. 6 bytes)[Note]
Medium model data pointer	Max. 4 bytes
Large model pointer	Max. 1 variable

Note Up to 6 bytes can be used only when the **-MS** and **-QF** options are specified.

Table 11-16. Restrictions on noauto Function Arguments and Automatic Variables (Without -ZO)

Data Type	Restriction
Type other than pointer	Max. 6 bytes (Max. 22 bytes)[Note 1]
Small model pointer Medium model data pointer	Max. 6 bytes (Max. 22 bytes)[Note 1]
Medium model function pointer	(Max. 5 variables)[Note 2]
Large model pointer	Max. 2 variables (Max. 7 variables)[Note 3]

Notes 1. When the **-QR** option is specified, only automatic variables can be used up to 22 bytes.

2. When the **-QR** option is specified, only automatic variables can be used up to 5 variables. The medium model function pointer cannot be used as a register argument (not allocated to registers).

3. When the **-QR** option is specified, only automatic variables can be used up to 7 variables.

- These restrictions are checked during compilation.
- If arguments and automatic variables are declared with a **register** (only when the **-ZO** is not specified), the **register** declaration is ignored.

noauto Function noauto

EXAMPLE

(C source)

```
noauto short nfunc (short, short, short);

short l, m;

void main (void)
{
   static short s1, s2, s3;
   l = nfunc (s1, s2, s3);
}

noauto short nfunc(short a, short b, short c)
{
   m = a + b + c;
   rturn(m);
}
```

(Output object of compiler) With small model, when **-ZO** option is not specified

```
@@DATA          DSEG
_l :            DS          (2)
_m :            DS          (2)
?L0003: DS      (2)
?L0004: DS      (2)
?L0005: DS      (2)

@@CODES CSEG    BASE
_main: s3
        push    ax
        movw    ax,!?L0004  ;s2
        push    ax
        movw    ax,!?L0003  ;s1
        call    !_nfunc     ;Calls nfunc (a, b, c)
        pop     ax,de
        movw    !_l,bc      ;Assigns return value to external variable l
        ret
```

noauto Function noauto

(Output object of compiler … continued)

```
_nfunc:
        push    rp3,vp,up       ;Saves register for arguments
        movw    rp3,ax          ;Assigns first argument a to RP3
        movw    ax,[sp+9]       ;Assigns second argument b to UP
        movw    up,ax           ;
        movw    ax,[sp+11]      ;Assigns third argument c to VP
        movw    vp,ax           ;
        movw    ax,rp3          ;To a (RP3)
        addw    ax,up           ;Adds b (UP)
        addw    ax,vp           ;Adds c (VP)
        movw    !_m,ax          ;Assigns the result of operation to external variable m
        movw    bc,ax           ;Returns external variable m
        pop     rp3,vp,up       ;Restores register for arguments
        g
```

EXPLANATION

- In the above example, the **noauto** attribute is added at the header part of the C source. **noauto** is declared and stack frame formation is not performed.

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the keyword **noauto** is not used.
- When changing variables to **noauto** variables, modify the program according to the method above.

From this C compiler to another C compiler

- **#define** must be used. For details, see **11.6 Modifications of C Source.**

(6) norec function

norec Function norec

FUNCTION

- A function that does not call another function by itself can be changed to a **norec** function.
- With the **norec** function, code for preprocessing and postprocessing (stack frame formation) is not output.
- All the arguments of **norec** function are allocated to registers and **saddr2** area (**_@NRARGX**) for arguments of the **norec** function. When the **-QR** option is not specified during compilation (default), however, **saddr2** area is not used.
- If arguments cannot be allocated to registers and **saddr2** area, a compilation error occurs.

(a) When -ZO option is specified

- Arguments are passed via a register and **saddr2** area (**_@NRARGX**). When a register is used, arguments are stored in the same manner as the **noauto** function (refer to **Table 11-13**).
- If arguments cannot be passed via a register, a register is not used, but arguments are passed via **saddr2** area (**_@NRARGX**) (a register and **saddr2** area are not used simultaneously).
  When **saddr2** area is used, arguments are sequentially stored in ascending order from **_@NRARG0** starting from the first argument.
- The locations where arguments are passed on the function call side and the function definition side become the locations where arguments are allocated.
- The save and restore of the register to which arguments are allocated are performed before/after the function call.
- Automatic variables are allocated to **saddr2** area (**_@NRATXX**), and so are the register variables. They are allocated in the sequence they have been declared in ascending order starting from **_@NRAT00**. If there are excess registers for arguments, automatic variables are allocated to registers first. However, automatic variables are allocated to **saddr2** area only when the **-QR** option is specified. If automatic variables cannot be allocated to registers or **saddr2** area, a compilation error occurs.
- The save and restore of the register to which automatic variables are allocated are performed on the function definition side.

norec Function norec

(b) When -ZO option is not specified

On the function call side, arguments are passed via a register and **saddr2** area (**_@NRARGX**) for the arguments of **norec** functions. On the function definition side, the arguments passed via a register are copied to a register (because the registers of the function call side and the function definition side are different). If arguments are passed via **saddr2** area, the location where arguments are passed becomes the location where arguments are allocated.

Arguments are allocated to registers first, and then the arguments that cannot be allocated to registers are allocated to **saddr2** area.

The save and restore of registers to store arguments are performed on the function definition side.

Automatic variables are allocated to registers or to **saddr2** area (**_@NRARGX**) for the arguments of the **norec** function if registers can be used. If the areas above cannot be used, automatic variables are allocated to **saddr2** area (**_@NRATXX**) for the automatic variables of the **norec** function in the sequence they have been declared and in ascending order.

The following shows the registers to be used for passing the arguments of **norec** functions.

Table 11-17. Registers Used for norec Function Arguments: Passing Side (Without -ZO)

Data Type	First Argument	Second Argument	Third Argument or Later
char	A	C	DE, RP2, saddr2 Note
int, short, enum	AX	DE	RP2, saddr2 Note
long/float/double/ long double	DE (higher 16 bits) AX (lower 16 bits)	saddr Note	saddr2
Small model pointer Medium model data pointer	AX	DE	RP2, saddr2 Note
Large model pointer	TDE	saddr2 Note	saddr2 Note

Note When the **-QR** option is specified, there arguments can be passed via **_@NRARGX (saddr2)**. Medium model function pointers (3 bytes) cannot be used as the arguments to be allocated to registers.

norec Function norec

Table 11-18. Registers Used for norec Function Arguments: Receiving Side (Without -ZO)

Data Type	First Argument	Second Argument	Third Argument or Later
char (with 4-byte arguments)[Note 1] char (without 4-byte arguments)[Note 1]	R10 R6	R11 R7	R6, R7, R8, R9, saddr2[Note 2] R10, R11, R8, R9, saddr2[Note 2]
int, short, enum (without 4-byte arguments)[Note 1] (with 4-byte arguments)[Note 1]	 UP RP3	 RP3 UP	 VP, saddr2[Note 2] VP, saddr2[Note 2]
long/float/double/long double	VP (higher16 bits) RP3 (lower 16 bits)	saddr2[Note 2]	saddr2[Note 2]
Small model pointer Medium model data pointer	UP	VP	RP3, saddr2[Note 2]
Large model pointer	VVP	saddr2[Note 2]	saddr2[Note 2]

Notes 1 4-byte arguments are arguments of **long**, **float**, **double** and **long double** type

2 When the **-QR** option is specified, these arguments can be passed via **_@NRARGX (saddr2)**. The medium model's function pointer (3 bytes) cannot be used as an argument assigned to the register.

Cautions 1. The medium model function pointers cannot be used as arguments to be allocated to registers.

2. The order of allocating registers of this function is the same as that of an ordinary function with the -QF option specified.

EFFECT

- The object code can be shortened and program execution speed can be improved.

USAGE

Declare a function with the **norec** attribute in the function declaration as follows.

```
norec type-name function-name
```

- **__leaf** can also be described instead of **norec**.

norec Function norec

RESTRICTIONS

- No other function can be called from a **norec** function.
- There are restrictions on the type and number of arguments and automatic variables that can be used in a **norec** function.
- When **-ZA** is specified, **norec** is disabled and only _ _**leaf** is enabled.
- The restrictions for arguments and automatic variables are checked during compilation, and an error occurs.
- If arguments and automatic variables are declared with a register, the register declaration is ignored.
- The following shows the types of arguments and automatic variables that can be used in **norec** functions.

- Pointer
- `char/signed char/unsigned char`
- `int/signed int/unsigned int`
- `short/signed short/unsigned short`
- `long/signed long/unsigned long`
- `float/double/long double`

(a) Restrictions for arguments of function when -ZO option is specified

- The **char** type arguments do not perform int extension.

Table 11-19. Restrictions on norec Function Arguments (When -ZO Is Specified)

Data Type	Restriction
char type	Max. 8 variables
int, **short**, small model pointer type	Max. 4 variables
Large model pointer, **long**, **float**, **double**, **long double** type	Max. 2 variables

norec Function norec

(b) Restrictions for arguments of function when -ZO option is not specified

Table 11-20. Restrictions on norec Function Arguments (When -ZO Is Not Specified)

Data Type	Restriction
Other than pointer	Max. 14 bytes (Max. 6 bytes)[Note]
Small model pointer, medium model data pointer	Max. 14 bytes (Max. 6 bytes)[Note]
Medium model function pointer	Max. 2 variables (cannot be used)[Note]
Large model pointer	Max. 3 variables (Max. 1 variable)[Note]

Note The figures enclosed in parentheses indicate values when **-QR** is not specified.

(c) Restrictions for automatic variables when -ZO option is specified

- Up to 8 bytes of the automatic variables can be used in the **norec** function.
  If there are excess registers used for arguments, they are added to the 8 bytes. Automatic variables are allocated to **saddr2** area in 1-byte alignment.
- In the case that the **-QR** option is not specified during compilation, if the total size of the arguments and automatic variables exceeds 4 bytes (6 bytes when **-MS -QF** is specified), an error occurs.

norec Function norec

(d) Restrictions for automatic variables when -ZO option is not specified

The automatic variables that can be used are allocated to the registers remaining after allocation of arguments, **saddr2** area (**_@NRARGX**) for the arguments of **norec** functions, and **saddr2** area (**_@NRATXX**) for automatic variables of **norec** functions.

Table 11-21. Restrictions on norec Function Automatic Variables (When -ZO Is Not Specified)

Data Type	Restriction
Other than pointer	Max. 22 bytes (Max. 6 bytes)[Note]
Small model pointer, medium model data pointer	Max. 22 bytes (Max. 6 bytes)[Note]
Medium model function pointer	Max. 4 variables (cannot be used)[Note]
Large model pointer	Max. 6 variables (Max. 2 variable)[Note]

Note The figures enclosed in parentheses indicate values when **-QR** is not specified.

EXAMPLE

(C source)

```
norec int rout (int a, int b, int c);

int i, j;
void main ( ) {
        int k, l, m;
        i = l + rout (k, l, m) + ++k ;
}

norec int rout (int a, int b, int c)
{
       int x, y;
       return (x + (a<<2));
}
```

norec Function norec

(Output object of compiler) (With large model, when **–QR** option is specified, and **-ZO** option is not specified)

```
EXTRN SADDR2 (_@NRARG0)        ;Refers to saddr2 area to be used.
PUBLIC        _rout
PUBLIC        _I
PUBLIC         _j
PUBLIC        _main

@@DATA DSEG
_i:    DS      (2)
_j:    DS      (2)

@@CODE CSEG
_main:
       push    uup
       subwg   sp, #06H
       movg    whl, sp
       movg    uup, whl
       movw    ax, [up+2]      ;Stores argument l to register RP2.
       movw    rp2, ax
       movw    ax, [up]        ;Stores argument m to register DE.
       movw    de, ax
       movw    ax, [up+4]      ;Stores argument k to register AX.
       call    $!_rout         ;Calls norec function
       movw    ax, [up+2]      ;Adds return value of norec function to l.
       addw    bc, ax          ;
       movw    ax, [up+4]      ;Increments k
       incw    ax              ;
       movw    [up+4], ax      ;
       addw    bc, ax          ;Assigns the result of addition to i.
       movw    !!_i, bc
       addwg   sp, #06H
       pop     uup
       ret
```

norec Function norec

(Output object of compiler…continued)

```
_rout:
        push    uup             Saves register for arguments.
        push    rp3             ;
        push    vvp             ;
        movw    rp3, ax         ;Assigns the first argument a to RP3.
        movw    vp, de          ;Assigns the third argument c to VP.
        movw    up, rp2         ;Assigns the second argument b to UP.
        movw    ax, rp3         ;Receives the first argument a from register RP3.
        shlw    ax, 2           ;
        addw    ax, _@NRARG0    ;Automatic variable x assigned to saddr2
        movw    bc, ax          ;Assigns return value to BC register
L0004:
        pop     vvp             ;Restores registers for arguments.
        pop     rp3
        pop     uup
        ret
        END
```

EXPLANATION

In the above example, the **norec** attribute is added in the definition of the **rout** function as well to indicate that the function is **norec**.

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the keyword **norec** is not used.
- When changing variables to **norec** variables, modify the program according to the method above.

From this C compiler to another C compiler

- **#define** must be used. For details, see **11.6 Modifications of C Source.**

(7) bit type variables

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 **saddr2** area.
- These variables can be handled the same as an external variable that has no initial value (or has an unknown value).
- The C compiler outputs the following bit manipulation instructions for these variables.

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

EFFECT

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

USAGE

- Declare a **bit** or **boolean** type inside the 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 the 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)

bit Type Variables
boolean Type Variables

bit
boolean
_ _boolean

RESTRICTIONS

- An operation on two **bit** or **boolean** type variables is performed by using the 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.
- The variable cannot be declared with an initial value.
- If the variable is described along with a **const** declaration, the **const** declaration is ignored.
- Only operations using 0 and 1 can be performed by the operators and constants shown in the following table.
- *, & (pointer reference, address reference), and **sizeof** operations cannot be performed.
- When the **-ZA** option is specified, only **_ _boolean** is enabled.

Table 11-22. Operators That Use Only Constants 0 or 1 (When Using bit Type Variable)

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

bit Type Variables
boolean Type Variables

bit
boolean
_ _boolean

Table 11-23. Number of Usable bit Type Variables

Condition	Restrictions (Per Load Module)
When **-QR** option is specified (**saddr2** area [XFD20H to XFDDFH])	Max. 1536 variables can be used.
When **-QR** option is not specified (**saddr2** area [XFD20H to XFDFFH])	Max. 1792 variables can be used.

The number of usable bit type variables is decreased if **sreg** variables are used or the **-RD** and **-RS** (automatic **saddr** allocation option) options are specified.

EXAMPLE

(C source)

```
#define ON   1
#define OFF  0

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

bit Type Variables
boolean Type Variables

bit
boolean
_ _boolean

(Assembler source)

Indicates the case where the user creates a definition code of a **bit** type variable. The following example shows the case of the large model (-ML) and the location 0FH (-CS15). In this example, if the compiler output section name **@@ BITS** is used, a link error occurs since the bit segment is changed to the **AT** attribute. Therefore, other segment names should be used (if the attribute is **saddr2**, the **@@BITS** segment name can be used).

```
PUBLIC _data1                    ;Declaration
PUBLIC _data2

BIT_SEG BSEG       AT 0FFD20H    ;Allocation to segment
_data1 DBIT
_data2 DBIT
```

(Output object of compiler)

If an **extern** declaration is not added, the compiler outputs the codes shown below. The following shows the case of the large model.

```
EXTRN  _testb
EXTRN  _chgb
PUBLIC _data1
PUBLIC _data2
PUBLIC _main

@@BITS   BSEG SADDR2
_data1 DBIT
_data2 DBIT
```

bit Type Variables **bit**
boolean Type Variables **boolean**
_ _boolean

(Output object of compiler…continued)

```
@@CODE CSEG
_main:
       set1    _data1          ;Initialize by 1
       clr1    _data2          ;Initialize by 0
L0003:
       bf      _data1, $L0004  ;Judgment
       mov1    CY, _data2      ;Assignment
       mov1    _data1, CY      ;Assignment
       call    !!_testb
       br      $L0003
L0004:
       bf      _data1, $L0005  ;Logical AND expression
       bf      _data2, $L0005  ;Logical AND expression
       call    !!_chgb
L0005:
L0006:
       ret
       END
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the keyword **bit**, **boolean**, or _ _**boolean** is not used.
- When changing variables to **bit** or **boolean** type variables, modify the program according to the method above.

From this C compiler to another C compiler

- **#define** must be used. For details, see **11.6 Modifications of C Source** (As a result of this modification, the **bit** or **boolean** type variables are handled as ordinary variables.).

(8) _ _boolean1 type variables

_ _boolean1 type variables _ _boolean1

FUNCTION

- _ _**boolean1** type variables are handled as 1-bit data and allocated to **saddr1** area.
- _ _**boolean1** type variables are handled in the same manner as external variables without an initial value (undefined).
- The compiler outputs the following bit manipulation instructions for these bit variables.

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

EFFECT

- Programming at the assembler source level and bit access to **saddr1** area are enabled by C description.

USAGE

- Declares _ _**boolean1** type in the module that uses _ _**boolean1** type variables.

`_ _boolean1` variable-name

- Declares the extern _ _**boolean1** in the module that refers to _ _**boolean1** type variables.

`extern _ _boolean1` variable-name

- The **sreg1** variables (except the element of an array and member of a union) of **char/int/short/long** types are automatically enabled to be used as _ _**boolean1** type variables.

variables-name`.n` (n is 0 to 31)

_ _boolean1 type variables _ _boolean1

RESTRICTIONS

- The operations between _ _**boolean1** type variables can be performed using carry flags. Therefore, the contents of the carry flag between statements are not guaranteed.
- _ _**boolean1** type variables cannot define/reference or array.
- _ _**boolean1** type variables cannot be used as a member of a structure or union.
- _ _**boolean1** type variables cannot be used as an argument type of a function.
- _ _**boolean1** type variables cannot be used as a return value of a function.
- _ _**boolean1** type variables cannot declare with an initial value.
- If described with the **const** declaration, the **const** declaration is ignored.
- Only operations using 0 and 1 can be performed by the operators and constants shown in the following table.
- *, & (pointer reference, address reference), and **sizeof** operations cannot be performed.

Table 11-24. Operators That Use Only Constants 0 or 1 (When Using bit Type Variables)

Classification	Operator	Category	Operator
Assignment	=		
AND in bit units	&, &=	OR in bit units	|, | =
XOR in bit units	^, ^		
Logical AND	&&	Logical OR	||
Equal	==	Not equal	!=

_ _boolean1 type variables _ _boolean1

The following shows the number of usable _ _**boolean1** type variables.

Table 11-25. Number of Usable _ _boolean1 Type Variables

Condition	Restrictions (Per Load Module)
When using **saddr1** area [XFE00H to XFE7FH]	Max. 1024 variables can be used.

When **sreg1** variables are used, however, the number of usable _ _**boolean1** type variables is decreased.

EXAMPLE

(C source)

```
#define ON       1
#define OFF      0

extern void testb (void);
extern void chgb (void);

extern _ _boolean1 data1;
extern_ _boolean1 data2 ;

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

_ _boolean1 type variables _ _boolean1

(Assembler source)

Indicates the case where the user creates a definition code of a _ _**boolean1** type variable. The following example shows the case of the large model (-ML) and the location 0FH (-CS15). In this example, if the compiler output section name **@@ BITS1** is used, a link error occurs since the bit segment is changed to an **AT** attribute. Therefore, other segment names should be used (if the attribute is **saddr**, the segment name **@@BITS1** can be used).

```
PUBLIC _data1                   ;Declaration
PUBLIC _data2

BIT1_SEG BSEG   AT 0FFE00H      ;Allocation to segment
_data1 DBIT
_data2 DBIT
```

(Output object of compiler)

The compiler outputs the following codes if an **extern** declaration is not added. The following shows the case of the large model.

```
EXTRN  _testb
EXTRN  _chgb
PUBLIC _data1
PUBLIC _data2
PUBLIC _main

@@BITS 1  BSEG  SADDR
_data1 DBIT
_data2 DBIT
```

_ _boolean1 type variables _ _boolean1

(Output object of compiler…continued)

```
@@CODE CSEG
_main :
       set1   _data1           ;Initialize by 1
       clr1   _data2           ;Initialize by 0
L0003 :
       bf     _data1, $L0004   ;Judgment
       mov1   CY, _data2       ;Assignment
       mov1   _data1, CY       ;Assignment
       call   !!_testb
       br     $L0003
L0004 :
       bf     _data1, $L0005   ;Logical AND expression
       bf     _data2, $L0005   ;Logical AND expression
       call   !!_chgb
L0005 :
L0006 :
       ret
       END
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the keyword _ _**boolean1** is not used.
- When changing to _ _**boolean1** type variables, modify the program according to the method above.

From this C compiler to another C compiler

- Changes are made by **#define**. For details, refer to **11.6 Modifications of C Source** (by these changes, _ _**boolean1** type variables are handled as ordinary variables).

(9) ASM statements

ASM Statements — #asm, #endasm _ _asm

FUNCTION

(a) #asm - #endasm

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

(b) _ _asm

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

EFFECT

- Global variables of the C source can be manipulated in the assembler source
- Functions that cannot be described in the C source can be implemented
- The assembler source output by the C compiler can be manually optimized and embeded it in the C source (to obtain efficient objects)

USAGE

(a) #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 Statements — #asm, #endasm _ _asm

(b) _ _asm

- Use of _ _**asm** is declared by the **#pragma asm** specification made at the beginning of the module in which the ASM statement is to be described (the uppercase letters and lowercase letters are distinguished for the keywords following **#pragma**).
- The following items can be described before **#pragma asm**.
  - Comment
  - Other **#pragma** directive
  - Preprocessing directive not creating variable definition/reference or function definition/reference
- The **ASM** statement is described in the following format in the C source.

```
_ _asm (string literal);
```

- The description method of the 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 the _ _**asm** block can only be described inside a function of the C source. Therefore, the assembler source is output to **CSEG** (with medium/large model) of the segment name **@@CODE** or **@@CODES CSEG BASE** (with small model).

ASM Statements

#asm, #endasm
_ _asm

EXAMPLE

(a) #asm - #endasm

(C source)

```
void main ( ) {
          #asm
                 callt  [60H]
          #endasm
}
```

(Output object of compiler)

The assembler source written by the user is output to the assembler source file.

```
@@CODE CSEG
_main:
                 callt  [60H]
                 ret
                 END
```

EXPLANATION

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

ASM Statements — #asm, #endasm _ _asm

(b) _ _asm

(C source)

```
#pragma             asm

int a, b;

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

(Assembler source)

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

COMPATIBILITY

- With a C compiler that 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.

(10) Interrupt functions

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) **saddr2** area for arguments/**auto** variables of **norec** function (regardless of whether the arguments or variables are used)

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 save/restore register contents, and that save/restore the contents of the **saddr2** area are not output regardless of whether the codes are used 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 during compilation, the **saddr2** area for register variables and the **saddr2** area for the arguments/**auto** variables of the **norec** function is not used; therefore, the saved/restored code is not output.
- If the size of the saved code is smaller than that of the restored code, the restored code is output.
- **Table 11-26** summarizes the above and shows the save/restore area.

Interrupt Functions

#pragma vect
#pragma interrupt

Table 11-26. Save/Restore Area When Interrupt Function Is Used

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	×	×	×	×	×	○	×	○	×
All registers	×	○	×	○	×	×	×	×	×
saddr2 area for register variable used	×	×	×	○	○	×	×	○	○
Entire saddr2 area for argument/auto variable of norec function	×	×	×	○	○	×	×	×	×

Stack: Use of stack is specified.
RBn: Register bank is specified.
○: Saved
×: Not saved

Caution If there is an ASM statement in an interrupt function, and if the area reserved for registers of the compiler is used in that ASM statement, the area must be saved by the user.

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.

Interrupt Functions — #pragma vect
#pragma interrupt

USAGE

- Specify an interrupt request name, a function name, stock switching registers, and whether the **saddr2** 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.
  - Comment statements
  - Preprocessing directive that neither defines nor references a variable or a function

```
#pragma Δ vect (or interrupt) Δ interrupt request name Δ function name Δ

        [stack change specification] Δ [ { stack use 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 char** (example: unsigned char buff [10];).
Stack use specification:	STACK (default)
No change specification:	NOBANK
Register bank specification:	RB0/RB1/RB2/RB3/RB4/RB5/RB6/RB7
Δ:	Space

Caution The startup routine of this compiler is initialized to register bank 0. Therefore, specifying register banks 1 to 7 is necessary.

Interrupt Functions

#pragma vect
#pragma interrupt

RESTRICTIONS

- An interrupt request name must be described in uppercase letters.
- A duplication check on interrupt request names will be made within only one module.
- If the same or another interrupt occurs because of the contents of the priority specification flag register and interrupt mask flag register while a vectored interrupt is being processed, the contents of the registers may be changed if a register bank is specified or no change is specified, resulting in an error. The compiler, however, cannot check this error.
- **callt/callf/noauto/norec/_ _callt/_ _callf/_ _leaf/_ _rtos_interrupt/_ _pascal/_ _flash** cannot be specified as the interrupt function.
- An interrupt function is specified with **void** type (example: **void** func (**void**);) because it cannot have an argument or a 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 on their own responsibility.
- If a function specifying no change, register bank, or stack change as the saving destination via a **#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.
- 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 a global **unsigned char** 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 the 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 compilation error occurs. It is possible to allocate an array in **saddr** area using 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.
- A stack change cannot be specified simultaneously with “no change”. If specified so, an error occurs.
- 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 occurs.

Interrupt Functions

#pragma vect
#pragma interrupt

EXAMPLE

(C source 1)

```
#pragma interrupt NMI inter rb1

void inter()
{
          /* interrupt handling to NMI pin input */
}
```

(Output object of compiler)

```
      @@BASE  CSEG  BASE
_inter:
      Register bank switching code
      Save code of saddr area for use by C compiler
      Interrupt handling to NMI input (function body)
      Restore code of saddr area for use by C compiler
      reti
      @@VECT02     CSEG    AT     02H ; NMI
              DW   _inter
```

(C source 2)

(When stack change and register bank are specified)

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

unsigned char buff[10];
void func();
void inter();
{
    func();
}
```

Interrupt Functions

#pragma vect
#pragma interrupt

(Output object of compiler) With large model

```
@@BASE  CSEG     BASE
_inter:
        sel      RB2             ;Changes register bank
        push     whl             ;
        movg     whl,sp          ;Changes stack pointer
        movg     sp,#_buff+10    ;
        push     whl             ;
        call     !!_func
        pop      whl             ;
        movg     sp,whl          ;Sets back stack pointer
        pop      whl             ;
        reti

@@VECT06     CSEG     AT 0006H
     DW       _inter
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if interrupt functions are not used at all.
- When changing an ordinary function to an interrupt function, modify the program according to the method above.

From this C compiler to another C compiler

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

(11) Interrupt function qualifier (_ _interrupt, _ _interrupt_brk)

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 functions.
- 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 functions.
- A function declared with this qualifier is regarded as a (non-maskable/maskable/software) interrupt function, and saves or restores the registers and variable areas (1) and (3) 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 arguments/**auto** variables of **norec** function (regardless of usage)

Remark If the **-QR** option is not specified (default) during compilation, codes to save or restore areas (2) and (3) are not output because these areas 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.

Interrupt Function Qualifier

_ _interrupt
_ _interrupt_brk

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

- **callt/callf/noauto/norec/_ _callt/_ _callf/_ _leaf/_ _rtos_interrupt/_ _pascal/_ _flash** cannot be specified for the interrupt function**.**

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 **11.5 (10) Interrupt functions**).

Interrupt Function Qualifier

_ _interrupt
_ _interrupt_brk

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
#pragma interrupt BRK_I inter_b RB2              /* Note */

_ _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 */
```

Note The interrupt request name of the software interrupt is “BRK_I.”

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required unless interrupt functions are supported.
- Modify the interrupt functions, if necessary, according to the method above.

From this C compiler to another C compiler

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

(12) Interrupt functions

Interrupt Functions — #pragma DI, #pragma EI

FUNCTIONS

- Codes **DI** and **EI** are output to an 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 for the declaration of an automatic variable, a comment, or a preprocessing 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 for comments and preprocessing directives), the **EI** code is output after the postprocessing of the function (immediately before the code **RET**).
- To output the **EI** code before the postprocessing of a function, close a new block after describing "**EI( )**;" (delimit this block with '}').

EFFECT

- A function disabling interrupts can be created.

USAGE

- Describe the **#pragma DI** and **#pragma EI** directives at the beginning of the C source. However, the following statement and directives may precede the **#pragma DI** and **#pragma EI** directives.
  - Comment statement
  - Other **#pragma** directives
  - Preprocessing directive that neither defines nor references a variable or function
- Describe **DI( )**; or **EI( )**; in the source in the same manner as a function call.
- **DI** and **EI** can be described in either uppercase or lowercase letters after **#pragma**.

Interrupt Functions — #pragma DI, #pragma EI

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

(C source 1)

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

(Output object of compiler)

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

Interrupt Functions

#pragma DI
#pragma EI

<To output **DI** after preprocessing and **EI** before postprocessing>

(C source 2)

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

(Output object of compiler)

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

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if interrupt functions are not used at all.
- When changing an ordinary function to an interrupt function, modify the program according to the method above.

From this C compiler to another C compiler

Delete the **#pragma DI** and **#pragma EI** directives or invalidate these directives by separating them with **#ifdef**. **DI** and **EI** can be used as ordinary function names (example: **#ifdef_ _K4_ _ ... #endif**).
When an ordinary function is to be used as an interrupt function, modify the program according to the specifications of each compiler.

(13) CPU control instruction

CPU Control Instructions #pragma HALT/STOP/BRK/NOP

FUNCTION

- The following codes are output to an object to create an object file.

(1) Instruction for HALT operation[Note 1]
(2) Instruction for STOP operation[Note 2]
(3) BRK instruction
(4) NOP instruction

Notes 1. The setting of STOP mode and selection of the internal system clock is possible using the STBC register. The C compiler reads STBC, checks the CK1/CK0 value of the internal system clock selection, and accordingly outputs the instruction to set the value for HALT to STBC.

2. The C compiler reads STBC, checks the CK1/CK0 value of the internal system clock selection, and accordingly outputs the instruction to set the value for STOP to STBC.

EFFECT

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

USAGE

- Describe the **#pragma HALT**, **#pragma STOP**, **#pragma NOP**, and **#pragma BRK** instructions at the beginning of the C source.
- The following items can be described before the **#pragma** directive.
  - Comment statement
  - Other **#pragma** directive
  - Preprocessing directive that neither defines nor references a variable or function
- The keywords following **#pragma** can be described in either uppercase or lowercase letters.

CPU Control Instructions #pragma HALT/STOP/BRK/NOP

- Describe as follows in uppercase letters in the C source in the same format as a 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()
{
        HALT();
        STOP();
        BRK();
        NOP();
}
```

CPU Control Instructions #pragma HALT/STOP/BRK/NOP

(Output object of compiler) With large model

```
@@CODE    CSEG
_main :
; line    7 :    HALT();
          mov    a,STBC
          bt     a,4,$$+12
          bt     a.5,$$+24
          mov    STBC,#01H
          br     $$+21
          bt     a.5,$$+9
          mov    STBC,#011H
          br     $$+12
          mov    STBC,#031H
          br     $$+6
          mov    STBC,#021H
; line    8 :             STOP() ;
          mov    a,STBC
          bt     a.4,$$+12
          bt     a.5,$$+24
          mov    STBC,#02H
          br     $$+21
          bt     a.5,$$+9
          mov    STBC,#012H
          br     $$+12
          mov    STBC,#032H
          br     $$+6
          mov    STBC,#022H
; line    9 :             BRK() ;
          brk
; line    10 :   NOP() ;
          nop
          ret
```

CPU Control Instructions #pragma HALT/STOP/BRK/NOP

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the CPU control instructions are not used.
- When the CPU control instructions are used, modify the program according to the method above.

From this C compiler to another C compiler

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

(14) callf functions

callf Functions — callf/_ _callf

FUNCTION

- The **callf** instruction stores the body of a function in the **callf** area. This makes code shorter than ordinary call instructions.
- If a function stored in the **callf** area is to be referenced without a prototype declaration, the function must be called by an ordinary call instruction.
- The callee (the function to be called) is the same as an ordinary function.

EFFECT

- The object code can be shortened.

USAGE

- Add the **callf** attribute or _ _**callf** attribute to the beginning of a function at the time of the function declaration as follows.

```
callf extern type-name function-name
_ _callf extern type-name function-name
```

RESTRICTIONS

- Functions declared with **callf** will be located in the **callf** entry area. At which address in the area each function is to be located will be determined at the time of linking object modules. For this reason, when using any **callf** function in an assembler source module, the routine to be created must be made “relocatable” using symbols.
- A check on the number of **callf** functions is made at linking time.
- **callf** entry area: 800H to FFFH
- The number of functions that can be declared with the **callf** attribute is not limited.
- The total number of functions with the **callf** attribute is not limited as long as the first function is within the range of [800H to FFFH].
- When the **-ZA** option is specified, only _ _**callf** is enabled.

callf Functions — callf/_ _callf

EXAMPLE

(C source 1)

```
_ _callf extern int fsub();

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

(C source 2)

```
_ _callf int fsub()

{
    int val;
    return val;
}
```

(Output object of compiler) With large model

```
<C source 1>
        EXTRN _fsub      ;Declaration
        Callf !_fsub     ;Call

<C source 2> (to be allocate to callf entry area)
        PUBLIC _fsub     ;Declaration

@@CALF  CSEG   FIXED
_fsub:                   ;Function definition
          :
        Function body
          :
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the keyword **callf/_ _callf** is not used.
- When changing functions to **callf** functions, modify the program according to the method above.

From this C compiler to another C compiler

- **#define** must be used to allow **callf** functions to be handled as ordinary functions.

(15) 16 MB expansion space utilization

16 MB Expansion Space Utilization

16 MB expansion -ML

FUNCTION

- An object file that can linearly access a 16 MB expansion space is created.

EFFECT

- The 16 MB expansion space can be accessed in the same manner as 16-bit addressing (64 KB) mode.

USAGE

- Specify the **-ML** option (default) during compilation.

RESTRICTIONS

- When the **-MS** option is specified at the time of startup:
  Small model: Combined code/data block capacity of 16 KB
- When the **-MM** option is specified at the time of startup:
  Medium model: Capacity of up to 1 MB for the code block and 16 KB for the data block
- When the **-ML** option is specified at the time of startup:
  Large model: Combined code/data block capacity of 16 MB, including up to 1 MB for the code block and 16 MB for the data block.

EXAMPLE

(C source)

```
sreg int  *ladr;
int  *grob;

void main ( ) {
        int atval;

        *ladr = atval;
        *grob = atval;
}
```

16 MB Expansion Space Utilization 16 MB expansion -ML

(Output object of compiler)
With small model

```
@@CODES      CSEG          BASE
_main :
       push   rp3           ;Preprocessing of function
       movw   ax,rp3
       movw   [_ladr],ax    ;*ladr = atval
       movw   hl,!_grob
       movw   ax,rp3
       movw   [hl],ax       ;*grob = atval
       pop    rp3           ;Postprocessing of function
       ret
```

With medium model

```
@@CODE CSEG
_main:
       push   rp3           ;Preprocessing of function
       movw   de,_ladr
       movw   ax,rp3
       movw   [de],ax       ;*ladr = atval
       movw   de,!!_grob
       movw   ax,rp3
       movw   [de],ax       ;*grob = atval
       pop    rp3           ;Postprocessing of function
       ret
```

16 MB Expansion Space Utilization 16 MB expansion -ML

(Output object of compiler)
With large model

```
@@CODE    CSEG
_main :
   push   rp3            ;Preprocessing of function
   movw   ax,rp3
   movw   [%_ladr],ax    ;*ladr = atval
   movg   whl,!!_grob
   movw   [hl],ax        ;*grob = atval
   pop    rp3            ;Postprocessing of function
   ret
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if it has been re-compiled with the **-ML** option added during compilation, when the 16 MB expansion space is to be used.

From this C compiler to another C compiler

- The source program need not be modified if it is re-compiled with each compiler.

(16) Allocation function

Allocation Function — Allocation function -CS

FUNCTION

- With the medium model (when the **-MM** option is specified) or with the large model (when the **-ML** option is specified), the allocation of the **saddr** area can be changed by using the **-CS** option.

EFFECT

- When the **-CS15** option is specified, the code space can be continuously used.

USAGE

- The **-CS** option is specified during compilation.

The **-CS** option performs the following operation.

-CS0:	Allocates **saddr** area to 0FD20H to 0FFFFH
-CS15/-CS0FH:	Allocates **saddr** area to 0FFD20H to 0FFFFFH
-CSA:	Does not check with compiler but with linker

RESTRICTIONS

- Use the startup routine included with to this compiler that specifies the location specified by the **-CS** option. The **LOCATION** instruction is described in the startup routine (for details of the startup routine, refer to the **CC78K4 C Compiler Operation User's Manual (U15557E)**).

EXAMPLE

(C source)

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

(Output object of compiler)

With large model (**-ML**) and location 0 (**-CS0**) specified

```
$CHGSFR (0)
$PROCESSOR(4026)
        ;Variable declaration etc.
@@CODE      CSEG
_main:
        ;Function preprocessing
        ;Function body processing
        ;Function postprocessing
        ret
```

Allocation Function — Allocation function -CS

With large model (**-ML**) and location 15 (**-CS15**) specified

```
$CHGSFR (15)
$PROCESSOR 4026)
        ;Variable declaration etc.
@@CODE      CSEG
_main:
        ;Function preprocessing
        ;Function body processing
        ;Function postprocessing
        ret
```

With large model (**-ML**) and without compile check (**-CSA**) specified

```
$CHGSFRA
$PROCESSOR(4026)
        ;Variable declaration etc.
@@CODE      CSEG
_main:
        ;Function preprocessing
        ;Function body processing
        ;Function postprocessing
      ret
```

COMPATIBILITY

From another C compiler to this C compiler

- When using the medium model or large model, modification is not required if the location position is specified by the **-CS** option during compilation and the source program is re-compiled.

From this C compiler to another C compiler

- The source program need not be modified if it is re-compiled with each compiler.

(17) Absolute address access function

Absolute Address Access Function #pragma access

FUNCTION

- A code to access the ordinary RAM space is output to the object with direct inline expansion, not by function call, and an object file is created.
- If the **#pragma** directive is not described, a function accessing an absolute address is regarded as an ordinary function.

EFFECT

- A specific address in the ordinary memory space can be easily accessed using C description.

USAGE

- Describe the **#pragma access** directive at the beginning of the C source.
- Describe the directive in the source in the same format as a function call.
- The following items can be described before **#pragma access**.
  - Comment statement
  - Other **#pragma** directives
  - Preprocessing directive that neither defines nor references a variable or function
- The keywords following **#pragma** can be described in either uppercase or lowercase letters.

The following four function names are available for absolute address accessing.

```
peekb, peekw, pokeb, pokew
```

Absolute Address Access Function #pragma access

[List of functions for absolute address accessing]

(a)
```
unsigned char peekb (addr);
unsigned int addr;  (small model)
unsigned long addr; (medium model/large model)
```

Returns 1-byte contents of address **addr**.

(b)
```
unsigned int peekw (addr);
unsigned int addr;  (small model)
unsigned long addr; (medium model/large model)
```

Returns 2-byte contents of address **addr**.

(c)
```
void pokeb (addr, data);
unsigned int addr;    (small model)
unsigned long addr;   (medium model/large model)
unsigned char data;
```

Writes 1-byte contents of data to the position indicated by address **addr**.

(d)
```
void pokew (addr, data);
unsigned int addr;    (small model)
unsigned long addr;   (medium model/large model)
unsigned int data;
```

Writes 2-byte contents of data to the position indicated by address **addr**.

RESTRICTIONS

- A function name for absolute address accessing must not be used.
- Describe functions for absolute address accessing in lowercase letters. Functions described in uppercase letters are handled as ordinary functions.

Absolute Address Access Function #pragma access

EXAMPLE

(C source)

```
#pragma access

char a;
int b;

void main ()
{
       a = peekb(0x1234);
       a = peekb(0xfe23);
       b = peekw(0x1256);
       b = peekw(0xfe68);

       pokeb(0x1234, 5);
       pokeb(0xfe23, 5);
       pokew(0x1256, 100);
       pokew(0xfe68, 100);
}
```

Absolute Address Access Function #pragma access

(Output object of compiler)
With large model

```
@@CODE CSEG
-main:
       mov     a, !01234H
       mov     !!_a,a
       mov     a, !0FE23H
       mov     !!_a,a
       movw    ax, !01256H
       movw    !!_b,ax
       movw    ax, 0FE68H
       movw    !!_b,ax
       mov     a, #05H ;5
       mov     !01234H,a
       mov     !0FE23H,a
       movw    ax, #064H      ;100
       movw    !01256H,ax
       movw    !0FE68H,ax
       ret
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if a function for absolute address accessing is not used.
- Modify the program according to the method above if a function for absolute address accessing is used.

From this compiler to another C compiler

- Delete the "**#pragma access**" statement or delimit with **#ifdef**. The function name of absolute address accessing can be used as a function name.
- When using a function for absolute address accessing, modify the program according to the specifications of each compiler (**#asm**, **#endasm**, **asm( );**,etc.)

(18) Bit field declaration

Bit Field Declaration — Bit field declaration

(1) Extension of type specifier

FUNCTION

- The bit field of **unsigned char** type is not allocated straddling over a byte boundary.
- The bit field of **unsigned int** type is not allocated straddling over a word boundary, but can be allocated straddling over a byte boundary.
- The bit fields of the same type are allocated in the same byte units (or word units). If the types are different, 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 specified, **unsigned char** 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;
```

Bit Field Declaration — Bit field declaration

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required.
- Change the type specifier to use **unsigned char** as the type specifier.

From this C compiler to another C compiler

- Modification is not required if **unsigned char** is not used as a type specifier.
- Change **unsigned char**, if it is used as a type specifier, to **unsigned int**.

(2) 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

- The **-RB** option is specified during compilation 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;
};
```

Bit Field Declaration — Bit field declaration

EXPLANATION

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

If the bit field is allocated to **saddr2** or **saddr1** area by the keywords **sreg/_ _sreg/_ _sreg1**, a bit manipulation instruction is output, and codes can be reduced.

Figure 11-1. Bit Allocation by Bit Field Declaration (Example 1)

Bit allocation from MSB with **-RB** option specified

MSB							LSB
a	b	c	d	e	f	g	h

Bit allocation from LSB without **-RB** option specified

MSB							LSB
h	g	f	e	d	c	b	a

EXAMPLE 2

(Bit field declaration)

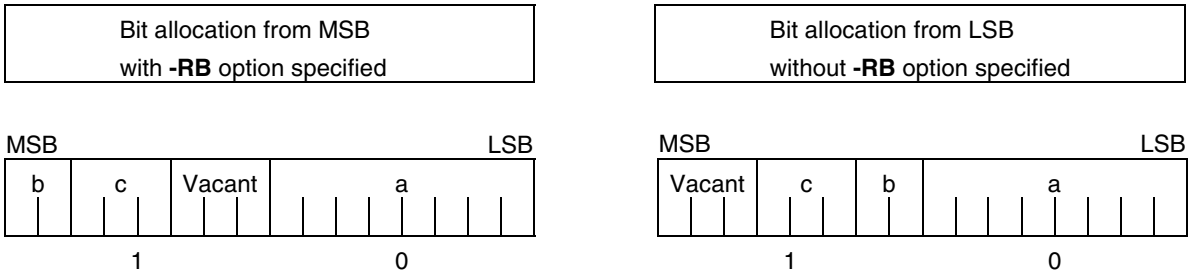
```
struct t {
       char         a;
       unsigned char  b:2;
       unsigned char  c:3;
       unsigned char  d:4;
       Int          e;
       unsigned int   f:5;
       unsigned int   g:6;
       unsigned char  h:2;
       unsigned int   i:2;
};
```

EXPLANATION

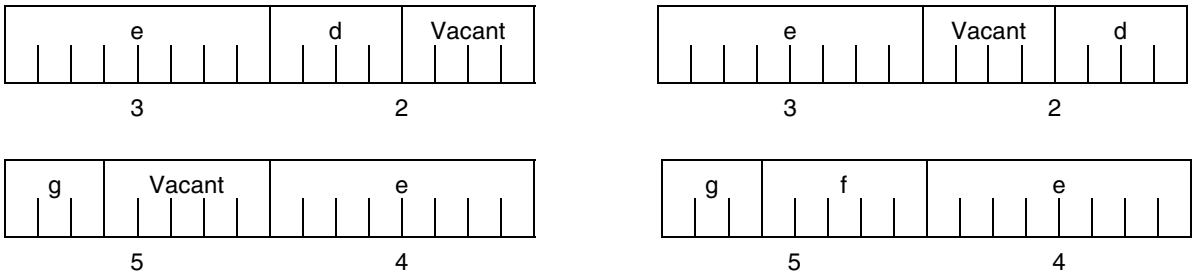
If the bit field is allocated to **saddr2** or **saddr1** area by the keywords **sreg/_ _sreg/_ _sreg1**, the code efficiency can be improved.

Bit Field Declaration — Bit field declaration

Figure 11-2. Bit Allocation by Bit Field Declaration (Example 2) (1/2)



Member **a** of **char** type is allocated to the first byte unit. **b** and **c** are allocated from the next byte unit. If the vacancy has run short, the members are allocated to the next byte unit. Because the vacancy is 3 bits and **d** is 4 bits in this example, **d** is allocated to the next byte unit.



The 78K/IV Series has 1-byte alignment; therefore, **e** (2 bytes) can straddle over a byte boundary.

Because **g** is an **unsigned** int type bit field, it can be allocated across byte boundary. **h** is an **unsigned char** type bit field; it is therefore allocated to the next byte unit, instead to the same byte unit as **g**, which is an **unsigned int** type bit field.

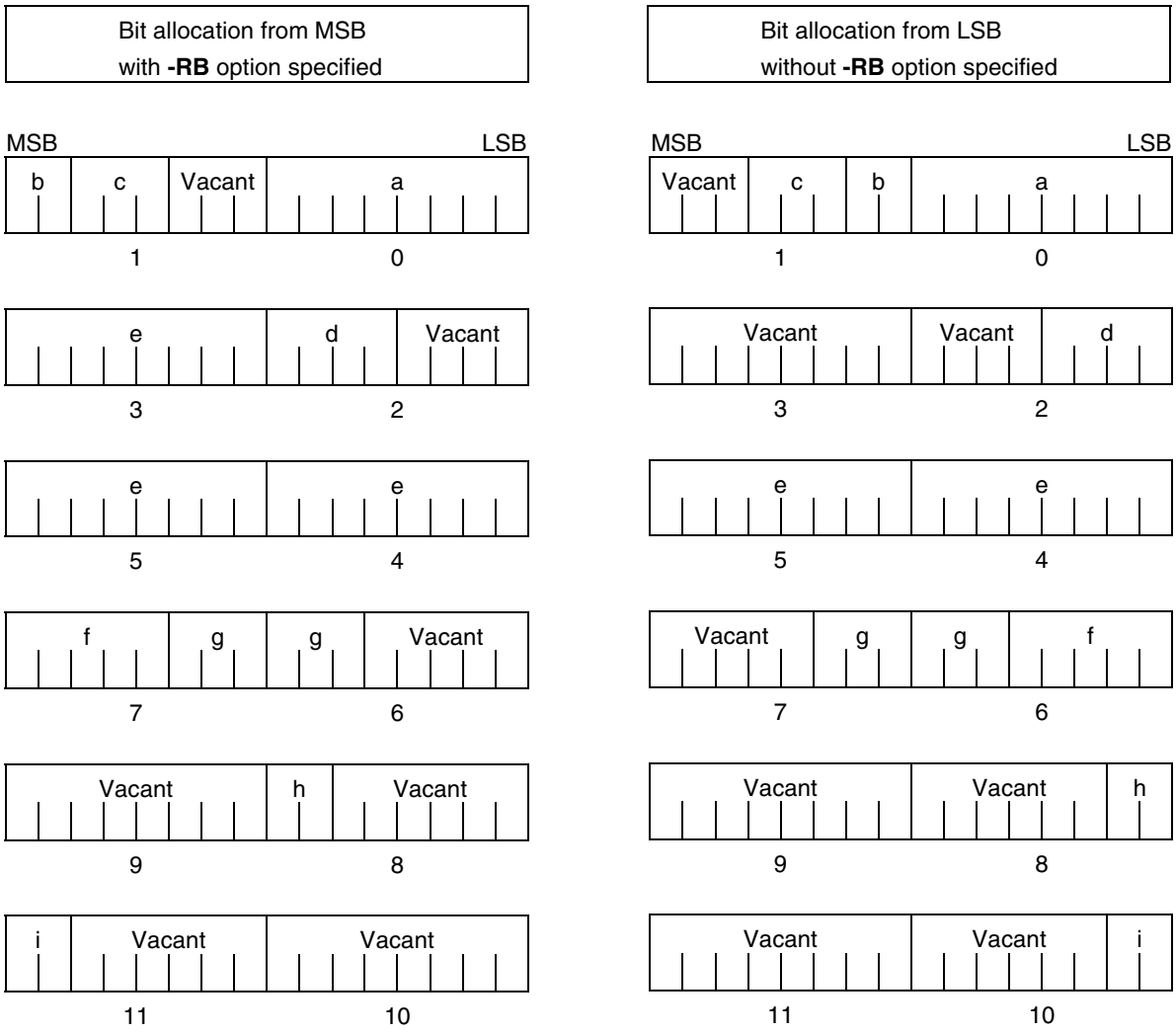
i is an **unsigned int** type bit field and can be allocated to the next word unit.

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

Bit Field Declaration — Bit field declaration

When the **-RA** option or **-RP** option is specified, the bit field is made 2-byte alignment. The location of the bit field above is as follows.

Figure 11-2. Bit Allocation by Bit Field Declaration (Example 2) (2/2)



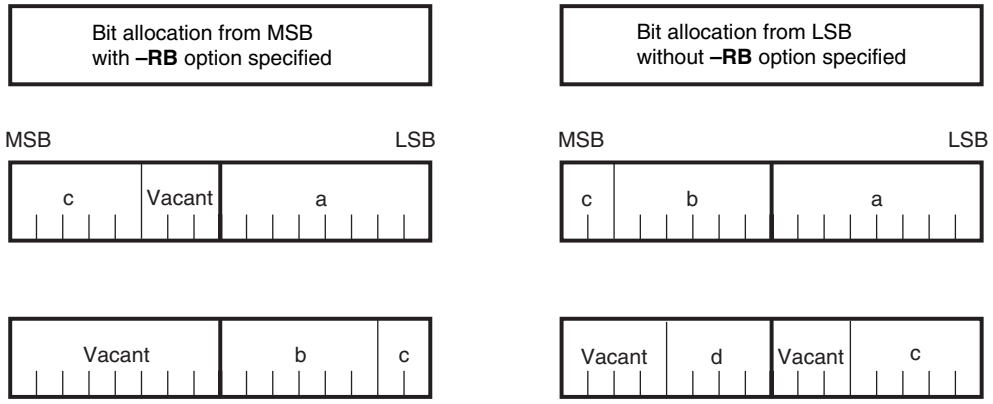
Bit Field Declaration — Bit field declaration

EXAMPLE 3

(Bit field declaration)

```
struct
   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;
};
```

Figure 11-3. Bit Allocation by Bit Field Declaration (Example 3) (1/2)

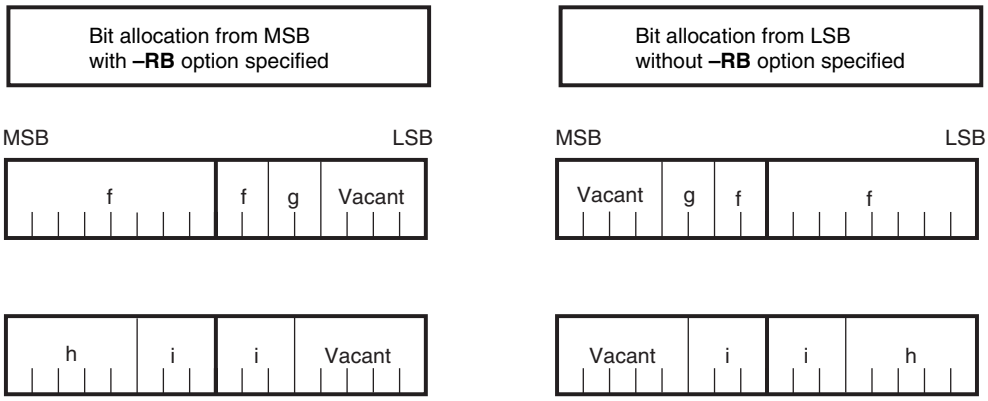


Since **b** and **c** are bit fields of type **unsigned int**, they are allocated from the next word unit.
Since **d** is also a bit field of type **unsigned int**, it is allocated from the next word unit.

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

Bit Field Declaration — Bit field declaration

Figure 11-3. Bit Allocation by Bit Field Declaration (Example 3) (2/2)



f and **g**, and **h** and **i** are each allocated to separate word units.

When the **–RA** option or **–RP** option is specified, the bit field is made 2-byte alignment. The location of the bit field above is as follows.

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

Bit Field Declaration — Bit field declaration

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required.

From this C compiler to another C compiler

- Modification is required if the **-RB** option is used and coding is performed taking the bit field allocation sequence into consideration.

(19) Changing compiler output section name

#pragma section… #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 **RA78K4 Assembler Package Operation User's Manual**.
- To change section names **@@CALT** and **@@CALF** with an **AT** start address specified, the **callt** and **callf** functions must be described before or after the other functions in the source file.
- If data is described after the **#pragma** directive is described, that data is located in the data change section. Another change instruction is possible, and if data is described after the rechange instruction, that data is located in the rechange section. If data defined before a change is redefined after the change, it is located in the rechanged section. Note that this is valid in the same way for **static** variables (within the function).

EFFECT

- Changing the compiler output section repeatedly in one file enables location of each section independently, so that data can be located independently in the desired data unit.

USAGE

- Specify the name of the section 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.
  - Comment statement
  - Preprocessing directive that neither defines nor references a variable or a function

However, all the sections in **BSEG** and **DSEG**, and the **@@CNST**, **@@CNSTS** and **@@CNSTM** sections 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 name.

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 a combination of these.

#pragma section… #pragma section

- The format in which the new section name is to be described conforms to the assembler specifications (up to eight 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 numbers, specify zero before F. It is therefore 0FFH.

- When the **-QR** option is not specified, the start address specification is within the following range.
  0XFE2C to 0XFE7F
- For sections other than the **@@CNST**, **@@CNSTS** and **@@CNSTM** sections in **CSEG**, that is, sections which locate functions, this **#pragma** directive cannot be described at other than the beginning of the C source (after the C text is described). If described, it causes an error.
- If this **#pragma** directive is executed after the C text is described, an assembler source file is created without an object module file being created.
- If this **#pragma** directive is described after the C text is described, a file that contains this **#pragma** directive and that does not have the C text (including external reference declarations for variables and functions) cannot be included. This results in an error (refer to **ERROR DESCRIPTION EXAMPLE 1**).
- An **#include** statement cannot be described in a file that executes this **#pragma** directive following the C text description. If described, it causes an error (refer to **ERROR DESCRIPTION EXAMPLE 2**).
- If **#include** statement follows the C text, this **#pragma** directive cannot be described after this description. If described, it causes an error. (Refer to **ERROR DESCRIPTION EXAMPLE 3**).

#pragma section… #pragma section

EXAMPLE 1

Section name **@@CODE** is changed to CC1 and address 2400H is specified as the start address.

(C source)

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

       void main()
       {
           Function body
}
```

(Output object)

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

EXAMPLE 2

This example shows a description in which this **#pragma** directive is described following the C text. The statement beginning with the double slashes (//) indicates the section to be located.

```
#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 = 2;                 // @@INIT and @@R_INIT
const int d2 = 2;                   // @@CNST
#pragma section @@DATA ??DATA2      // ??DATA is closed automatically 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… #pragma section

```
#pragma section @@DATA @@DATA          // ??DATA2 is closed and processing returns to the default
                                       // @@DATA.
#pragma section @@INIT ??INIT
#pragma section @@R_INIT ??R_INIT
                                       //If both names @@INIT and @@R_INIT are not changed,
                                       // ROMization becomes invalid.
        int a4;                        // @@DATA
        _sreg int b4;                  // ??DATS
        int c4 = 4;                    // ??INIT and ??R_INIT
const int d4 = 4;                      // @@CNST
#pragma section @@INIT @@INIT
#pragma section @@R_INIT @@R_INIT
                                       // ??INIT and ??R_INIT are closed and return to the defaults
#pragma section @@BITS ??BITS
        _boolean e4;                   // ??BITS
#pragma section @@CNST ??CNST
        char*const p = "Hello";        // both p and "Hello" ??CNST
```

EXAMPLE 3

```
#pragma section @@INIT ??INIT1
#pragma section @@R_INIT ??R_INT1
#pragma section @@DATA ??DATA1
        char c1;
        int i2;
#pragma section @@INIT ??INIT2
#pragma section @@R_INIT ??R_INT2
#pragma section @@DATA ??DATA2
        char c1;
        int i2 = 1;
#pragma section @@DATA ??DATA3
#pragma section @@INIT ??INIT3
#pragma section @@R_INIT ??R_INT3
        extern char c1;                // ??DATA3
        int i2;                        // ??INIT3 and ??R_INT3
#pragma section @@DATA ??DATA4
#pragma section @@INIT ??INIT4
#pragma section @@R_INIT ??R_INT4
```

#pragma section… #pragma section

EXAMPLE 4

(Method to specify the location of a section by link directives)

1. Change the section name whose location is to be changed in the C source.
   (In this example, **@@DATA** is changed to **DAT1**, and **@@INIT** is changed to **DAT2**)

(C source)

```
#pragma section @@DATA  DAT1
#pragma section @@INIT  DAT2

unsigned int d1,d2,d3;
unsigned long l1, l2;
unsigned int i =1;
        :
```

(Output object of compiler)

```
@@R_INT        CSEG          ;
       DW      01H           ;1

DAT2   DSEG
_I :   DS      (2)

DAT1   DSEG
_d1 :  DS      (2)
_d2 :  DS      (2)
_d3:   DS      (2)
_l1 :  DS      (4)
_l2 :  DS      (4)
```

2. Create a link directive file.

(Link directive file lk78k4.job)

```
memory EXTRAM1:(0F0000h , 01000h)
memory EXTRAM2:(0F1000h , 01000h)
   :
merge DAT1 : = EXTRAM1
merge DAT2 : AT(0F1000h) = EXTRAM2
```

#pragma section… #pragma section

3. Link by specifying the link directive file using the linker option **-D**.

```
> lk78k4  s4.rel  sample.rel  -BCl4.lib  -Dlk78k4.job  -S
```

The following example explains the restrictions on describing this **#pragma** directive following the C text.

ERROR DESCRIPTION EXAMPLE 1

```
a1.h
    #pragma section @@DATA ??DATA1    // File with a #pragma section only.

a2.h
    extern int func1 (void);
    #pragma section @@DATA ??DATA2    // File where there is C text and this #pragma directive follows
                                      // after.

a3.h
    #pragma section @@DATA ??DATA3    // File with a #pragma section only.

a4.h
    #pragma section @@DATA ??DATA3
    extern int func2 (void);          // File that includes C text.

a.c
    #include "a1.h"
    #include "a2.h"
    #include "a3.h"                   // ← Error.
                                      // There is C text in a2.h and after that this #pragma directive is
                                      // included, so the file that includes this #pragma directive only, //
                                      a3.h, cannot be included.
    #include "a4.h"
```

#pragma section… #pragma section

ERROR DESCRIPTION EXAMPLE 2

```
b1.h
    const int i;

b2.h
    const int j;
    #include "b1.h"                 // There is C text and there is no file (b.c) where this #pragma
                                    // directive is executed after it, so there is no error.
b.c
    const int k;
    #pragma section @@DATA ??DATA1
    #include "b2.h"                 // ←Error.
                                    // There is C text, and in the file following it where this #pragma
                                    // directive is executed (b.c), a subsequent #include statement
                                    // cannot be described.
```

ERROR DESCRIPTION EXAMPLE 3

```
c1.h
    extern int j;
    #pragma section @@DATA ??DATA1  // This #pragma directive is included and processed before c3.h
                                    processing, so there is
                                    // no error.
c2.h
    extern int k;
    #pragma section @@DATA ??DATA2  // ← Error.
                                    // There is C text in c3.h and after that there is an #include
                                    // statement, so this #pragma directive cannot be included after
                                    // that.
c3.h
    #include "c1.h"
    extern int i;
    #include "c2.h"
    #pragma section @@DATA ??DATA3  // ← Error.
                                    // There is C text, and after that there is an #include statement, so
                                    // this #pragma directive cannot be included after that.
```

#pragma section… #pragma section

```
c.c
    #include "c3.h"
    #pragma section @@DATA??DATA4     // ← Error.
                                      // There is C text in c3.h and after that there is an #include
                                      // statement, so this #pragma directive cannot be included after
                                      // that.
    int i;
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the section name change function is not supported.
- When changing the section name, modify the program according to the method above.

From this C compiler to another C compiler

- Delete **#pragma section** ... or delimit it with **#ifdef**.
- When changing the section name, modify the program according to the specifications of each compiler.

RESTRICTIONS

- A section name that indicates a segment for the vector table (e.g., **@@VECT02**) 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 occurs.
- When changing compiler output section names **@@DATS**, **@@BITS**, and **@@INIS**, limit the range of the specified address within **saddr2** area.
  (**saddr2** area)
  0xFD20 to 0xFDFF (With the small model, or when **-CS0** of the medium model/large model is specified)
  0xFFD20 to 0xFFDFF (When **-CS15** of the medium model/large model is specified or default)
- When changing compiler output section names **@@DATS1**, **@@BITS1**, and **@@INIS1**, limit the range of the specified address within **saddr1** area.
  (**saddr1** area)
  0xFE00 to 0xFEFF (With the small model, or when **-CS0** of the medium model/large model is specified)
  0xFFE00 to 0xFFEFF (When **-CS15** of the medium model/large model is specified or default)

  Remark Of the areas shown above, 0xXFE80 to 0xXFEFF (When **-CS0** is specified: X = 0, when **-CS15** is specified: X = F) are areas for registers. Care must be taken when specifying these areas.

- When the **-CSA** option is specified, the following addresses cannot be specified for the start address specification.
  0xFD00 to 0xFEFF, 0xFFD00 to 0xFFEFF

#pragma section… #pragma section…

CAUTION

- A section is equivalent to a segment of the assembler.
- The compiler does not check whether the new section name is duplicated with another symbol. Therefore, the user must check that the section name is not duplicated by assembling the output assemble list.
- If a section name (*) related to ROMization is changed by using **#pragma section**, the startup routine must be changed by the user on his/her own responsibility.

(*) ROMization-related section name

```
@@R_INIT, @@R_INIS, @@RSINIT, @@RSINIS
@@INIT, @@INIS, @@RSINS1, @@R_INS1, @@INIS1
```

The startup routine to be used when a section related to ROMization is changed, and an example of changing the end module are described below.

[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 end module (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 in the new section.
Therefore, add the declaration of the first label of the new section and the portion that copies the initial value to the startup routine, and add the portion that declares the end label to the end module, 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**.

(Changing startup routine cstartx.asm)

(1) Add the declaration of the end label of the section whose name has been changed.

```
EXTRN     _main, _@STBEG, _hdwinit
EXTRN     RTT1_E, TT1_E ← Add EXTRN declaration of RTT1_E, TT1_E
```

(2) Add the portion that copies the initial value from the **RTT1** section whose name has been changed to the **TT1** section.

The initial value copying processing codes differ depending on the memory model. Initial value copying processing can easily be added by copying the corresponding portion (initial value copying processing code) from the startup routine referring to the memory model specified by **$_IF**, changing the symbols of the changed section **_@R_INIT**, **_?R_INIT**, etc. to **RTT1_S**, **RTT1_E**, etc., and adding the changed branch symbol (to **LTT1**, etc.).

```
          :
          MOV     [DE+],A
          BR      $LDATS11
LDATS12 :

; RTT1-> part added with TT1 copying processing (start)

          MOVG    TDE,#TT1_S
          MOVG    WHL,#RTT1_S
LTT1 :
          SUBG    WHL,#RTT1_E
          BE      $LTT2
          ADDG    WHL,#RTT1_E
          MOV     A,[HL+]
          MOV     [DE+],A
          BR      $LTT1
LTT2 :

; RTT1 -> part added with TT1 copying processing (end)

$_IF(SMALL)
          CALL    !_main        ;main();
$ELSE
          CALL    !!_main       ;main();
$ENDIF
          BR      $$
```

Add portion that copies initial value from RTT1 section to TT1 section

(3) Set the first label of the section whose name has been changed. For the attribute of segment, refer to **APPENDIX B LIST OF SEGMENT NAMES**.

```
        :
$_IF(SMALL)
@@RSINS1      CSEG   BASE
$ELSE
@@R_INS1      CSEG
$ENDIF
_@R_INS1:
@@INIS1       DSEG   SADDR
_@INIS1:
@@DATS1       DSEG   SADDR
_@DATS1:
```

```
RTT1     CSEG
RTT1_S:  Add setting of label indicating beginning of section RTT1

TT1      DSEG
TT1_S:   Add setting of label indicating beginning of section TT1
```

```
$_IF(SMALL)     BASE
@@CALFS CSEG    FIXEDA
@@CNSTS CSEG    BASE
$ENDIF
$_IF(MEDIUM)
@@CODE  CSEG
        :
;
END
```

#pragma section… #pragma section…

(Changing end module rom.asm)

(1) Declare the label indicating the end of the section whose name has been changed.

```
       :
$ELSE
       NAME      @rom
$ENDIF
       :
PUBLIC _?R_INIT,_?R_INIS
PUBLIC _?INIT,_?DATA,_?INIS,_?DATS
PUBLIC _?R_INS1,_?INIS1,_?DATS1

PUBLIC RTT1_E, TT1_E   ← Add RTT1_E and TT1_E

;
$ELSE
@@INIT  DSEG
_?INIT:
@@DATA         DSEG
_?DATA:
$ENDIF
@@INIS  DSEG   SADDR2
_?INIS:
@@DATS  DSEG   SADDR2
_?DATS:
@@R_INS1       CSEG
_?R_INS1:
@@INIS1 DSEG   SADDR
_?INIS1:
@@DATS1 DSEG   SADDR
_?DATS1:
$ENDIF
;
```

(2) Set the label indicating the ends.

```
            :
RTT1    CSEG        Add setting of label indicating end of section RTT1
RTT1_E:

TT1     DSEG        Add setting of label indicating end of section TT1
TT1_E:
 ;
          END
```

(20) Binary constant

Binary Constant — Binary constant 0bxxx

FUNCTION

- Describes binary constants at the location where integer constants can be described.

EFFECT

- Constants can be described in bit strings without being replaced with octal or hexadecimal numbers. Readability is also improved.

USAGE

- Describe binary constants in the C source. The following shows the description method for 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.
  - Non-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: **unsigned long int**

Binary Constant — Binary constant 0bxxx

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 this C compiler

- Modifications are not needed.

From this C compiler to another C compiler

- Modification is required to meet the specifications 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.

(21) Module name changing function

Module Name Changing Function #pragma name

FUNCTION

- Outputs the first eight letters of the specified module name to the symbol information table in an object module file.
- Outputs the first eight letters of the specified module name to the assemble list file as symbol information (**MOD_NAM**) when **–G2** is specified and as the **NAME** quasi directive when **-NG** is specified.
- If a module name with nine or more letters is specified, a warning message is output.
- If unauthorized letters are described, an error occurs and the processing is aborted.
- If more than one of this **#pragma** directive exists, a warning message is output, and whichever is 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 '(' ')'. Upper case and lowercase letters are distinguished.

EXAMPLE

```
#pragma name module1
          :
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the compiler does not support the module name changing function.
- When changing a module name, modify the program according to the method above.

From this C compiler to another C compiler

- Delete **#pragma** name … or delimit it with **#ifdef**.
- When changing a module name, modify the program according to the specification of each compiler.

(22) Rotate function

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

- The rotate function can be realized using C source or **ASM** description without describing the processing to perform rotate.

USAGE

- Describe in the source in the same format as a function call. There are the following four 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 the right y times.

(b)
```
unsigned char rolb (x, y) ;
unsigned char x ;
unsigned char y ;
```

Rotates x to the left y times.

(c)
```
unsigned int rorw (x, y) ;
unsigned int x ;
unsigned char y ;
```

Rotates x to the right y times.

(d)
```
unsigned int rolw (x, y)
unsigned int x ;
unsigned char y ;
```

Rotates x to the left y times.

Rotate Function #pragma rot

- Declare the use of the function for rotate by the **#pragma rot** directive of the module.
  However, the following items can be described before **#pragma rot**.
  - Comments
  - Other **#pragma** directives
  - Preprocessing directives that neither define nor reference variables or 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 ( ) {
   c = rorb(a, b);
}
```

(Output assembler source) with large model

```
_main:
      mov    c,!!_b
      mov    a,!!_a
      ror    a,1
      dbnz   c,$$-2
      mov    !!_c,a
      ret
```

Rotate Function #pragma rot

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 this C compiler

- Modification is not required if the compiler does not use the functions for rotate.
- When changing to functions for rotate, modify the program according to the method above.

From this C compiler to another C compiler

- Delete the **#pragma rot** statement or delimit it with **#ifdef**.
- When using as a function for rotate, modification is required according to the specification of each compiler (**#asm**, **#endasm** or **asm()** ; , etc.).

(23) Multiplication function

Multiplication Function #pragma mul

FUNCTION

- Outputs the code that multiplies the value of an expression to an 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

- Codes utilizing the data size of the multiplication instruction I/O are generated. Therefore, 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 a function call in the source. There are the following three functions for multiplication.

```
mulu, muluw, mulw
```

[List of multiplication functions]

(a)
```
unsigned int mulu (x, y);
unsigned char x;
unsigned char y;
```

Performs unsigned multiplication of x and y.

(b)
```
unsigned long muluw (x, y);
unsigned int x;
unsigned int y;
```

Performs unsigned multiplication of x and y.

(c)
```
long mulw (x, y);
int x;
int y;
```

Performs signed multiplication of x and y.

Multiplication Function #pragma mul

- Declare the use of functions for multiplication with the **#pragma mul** directive of the module.
  However, the following items can be described before **#pragma mul**.
  - Comments
  - Other **#pragma** directives
  - Preprocessing directives that neither define nor reference variables or functions.
- Keywords following **#pragma** can be described in either uppercase or lowercase letters.

RESTRICTIONS

- Multiplication functions are handled as ordinary function if the target device does not have multiplication instructions.
- The function names for multiplication cannot be used as the function names (when **#pragma mul** is declared).
- The functions for multiplication must be described in lowercase letters. If they are described in uppercase letters, they are handled as ordinary function.

EXAMPLE

(C source)

```
#pragma mul
unsigned char a = 0x11;
unsigned char b = 2;
unsigned int I;
void main()
{
       i = mulu(a, b);
}
```

(Output object of compiler)

```
_main:
      mov    a,!!_b
      mov    b,!!_a
      mulu   b
      movw   !!_i,ax
      ret
```

Multiplication Function #pragma mul

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the compiler does not use the functions for multiplication.
- When changing to functions for multiplication, modify the program according to the method above.

From this C compiler to another C compiler

- Delete the **#pragma mul** statement or delimit it with **#ifdef**. Function names for multiplication can be used as the function names.
- When using as functions for multiplication, modification is required according to the specification of each compiler (**#asm**, **#endasm** or **asm()** ;, etc.).

(24) Division function

Division Function #pragma div

FUNCTION

- Outputs the code that divides the value of an expression to an object with direct inline expansion instead of function call and generates an object code file.
- If there is not a **#pragma** directive, the function for division is regarded as an ordinary function.

EFFECT

- Codes utilizing 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 a function call in the source. There are the following two functions for division.

```
divuw, moduw
```

[List of division functions]

(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 functions for division with the **#pragma div** directive of the module.
  However, the following items can be described before **#pragma div**.
  - Comments
  - Other **#pragma** directives
  - Preprocessing directives that neither define nor reference variables or functions.
- Keywords following **#pragma** can be described in either uppercase or lowercase letters.

Division Function #pragma div

RESTRICTIONS

- The division function is handled as an ordinary function if the target device does not have division instructions.
- 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 () {
       i = divuw(a, b);
       c = moduw(a, b);
}
```

(Output object of compiler) With large model

```
_main:
       mov     b,!!_b
       movw    ax,!!_a
       divuw   b
       movw    !!_i,ax
       mov     b,!!_b
       movw    ax,!!_a
       divuw   b
       mov     !!_c,b
       ret
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the compiler does not use the functions for division.
- When changing to functions for division, modify the program according to the method above.

From this C compiler to another C compiler

- Delete the **#pragma div** statement or delimit it with **#ifdef**. The function names for division can be used as the function names.
- When using as a function for division, modification is required according to the specification of each compiler (**#asm**, **#endasm** or **asm()** ; , etc.).

(25) Data insertion function

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 instructions can be embedded in the code area without using the **ASM** statement.
  When **ASM** is used, an object cannot be obtained without going through the assembler. On the other hand, if the data insertion function is used, an object can be obtained without going through the assembler.

USAGE

- Describe using uppercase letters in the source in the same format as that of a function call.
- The function name for data insertion is _ _**OPC**.

[List of data insertion functions]

(a) `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 with the **#pragma opc** directive.
  However, the following items can be described before **#pragma opc**.
  - Comments
  - Other **#pragma** directives
  - Preprocessing directives that neither define nor reference variables or functions.
- Keywords following **#pragma** can be described in either uppercase or lowercase letters.

Data Insertion Function #pragma opc

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 it is described in lowercase letters, it is handled as an ordinary function.

EXAMPLE

(C source)

```
#pragma opc
void main ( ) {
      _ _OPC(0xBF);
      _ _OPC(0xA1, 0x12);
      _ _OPC(0x10, 0x34, 0x12);
}
```

(Output object of compiler)

```
 _main:
 ; line   4 : _ _OPC (0xBF);
       DB    0BFH
 ; line   5 : _ _OPC (0xA1, 0x12);
       DB    0A1H
       DB    012H
 ; line   6 : _ _OPC (0x10, 0x34, 0x12);
       DB    010H
       DB    034H
       DB    012H
       ret
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the compiler does not use the functions for data insertion.
- When changing to functions for data insertion, modify the program according to the method above.

From this C compiler to another C compiler

- Delete the **#pragma opc** statement or delimit it with **#ifdef**.
- Function names for data insertion can be used as function names. When using as a function for data insertion, modification is required according to the specification of each compiler (**#asm**, **#endasm** or **asm()** ; , etc.).

(26) Interrupt handler for real-time OS (RTOS)

Interrupt Handler for RTOS #pragma rtos_interrupt ...

FUNCTION

- Interprets the function name specified by the **#pragma rtos_interrupt** directive as the interrupt handler for the 78K/IV Series RTOS (real-time OS) RX78K/IV.
- Registers the address of the described function name to the interrupt vector table for the specified interrupt request name.
- When a stack change is specified, the stack pointer is changed to the location where the 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 a global **unsigned char** type array.

  The two system call calling functions **ret_int/ret_wup** can be called in the interrupt handler for RTOS (for the details of the system call calling function, refer to the **List of RTOS System Call Calling Functions** described later).

  If the prototype declaration or the entity definition of **ret_int/ret_wup** and **ret_int/ret_wup** are called outside the interrupt handler for RTOS, an error occurs.

  The two RTOS system call calling functions **ret_int/ret_wup** are called by an unconditional branch instruction.

  If there is neither **ret_int** nor **ret_wup** in the interrupt handler for RTOS, an error occurs.

  If the interrupt request name and thereafter is omitted, only the two functions **ret_int/ret_wup** are enabled.

  The interrupt handler for RTOS generates codes in the following order.

(1) Saves all the registers
(2) Changes the stack pointer (only when stack change is specified)
(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) Sets back the stack pointer (only when stack change is specified)
(7) Restores all the registers
(8) reti

For **ret_int/ret_wup** described in the middle of the function, the codes in (5) and (6) are generated immediately before the unconditional branch instruction each time.

If a function ends with **ret_int/ret_wup**, the codes in (7) and (8) are not generated.

Interrupt Handler for RTOS #pragma rtos_interrupt ...

EFFECT

- The interrupt handler for RTOS can be described at 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, and stack change is specified by the **#pragma** directive.
- This **#pragma** directive is described at the start of the C source.
  When **#pragma PC** (type) is described, the main **#pragma** directive is described after **#pragma PC**.
  The following items can be described before **#pragma** directive.
  - Comments
  - Preprocessing directives that neither define nor reference variables or functions.

```
#pragmaΔrtos_interrupt [Δ Interrupt request name Δ function name Δ [stack change specification]]
```

Remark Stack change specification: SP = array name [+ offset location]

- 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.

[List of RTOS system call calling functions]

(1)
```
void ret_int ( );
```
Calls RTOS system call ret_int.

(2)
```
void ret_wup (x);
char *x;
```

Calls RTOS system call ret_wup with x as an argument.

Interrupt Handler for RTOS #pragma rtos_interrupt ...

RESTRICTIONS

- 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 occurs.
- A duplication check on interrupt request names will be made within only one module.
- If an interrupt (the same or another interrupt) is generated in duplicate during vector interrupt processing due to the contents of the priority specification flag register, interrupt mask flag register, etc., if the stack change is specified, the contents of the stack are updated, which may cause problems. However, this cannot checked by the compiler, so care must be taken.
- **callt/callf/noauto/norec/_ _callt/_ _callf/_ _leaf/_ _interrupt/_ _interrupt_brk/_ _pascal/_ _flash** cannot be specified for the interrupt handler for RTOS.
  The RTOS system call calling function names **ret_int/ret_wup** cannot be used for the function names.
  If the functions that specified the stack change via the **#pragma rtos_interrupt** specification are not defined in the same module, a warning is output and the stack change specification is ignored.
  The interrupt handler for RTOS is not supported when the static model is specified.

EXAMPLE

(a) When stack change is not specified

(C source)

```
#pragma rtos_interrupt  INTP0  intp
int  I;
void  intp ( ) {
   int a;
   a = 1;
   if (i == 1) {
          ret_int();
   }
}
```

Interrupt Handler for RTOS #pragma rtos_interrupt ...

(Output object of compiler)
When **-ML**, **-QV** is specified (default)

```
@@BASE CSEG    BASE
_intp:
       push    whl             ;Saves register
       push    tde
       push    uup
       push    vvp
       push    ax,bc,rp2,rp3
       movw    rp3,#01H        ;Allocates RP3 to variable a Note
       movw    ax,!!_i
       cmpw    ax,rp3
       bne     $L0003
       br      !!_ret_int
L0003;
       pop     ax,bc,rp2,rp3   ;Restores register
       pop     vvp
       pop     uup
       pop     tde
       pop     whl
       reti

@@VECT06       CSEG    AT     0006H
_@vect06:
       DW      _intp
```

Note When the **-QV** option is not specified, the securing/releasing codes of the local variables are output after saving the register/before restoring the register, respectively.

Interrupt Handler for RTOS #pragma rtos_interrupt ...

(b) When the stack change is specified

(C source)

```
#pragma rtos_interrupt INTP0 intp sp=buff+10
int I;
unsigned char buff[10];
extern unsigned short  TaskID1;
void intp () {
       int a;
       a = 1;
       if (i == 1) {
            ret_wup (&TaskID1);
       }
}
```

(Output object of compiler)
When **-ML**, **-QV** is specified (default)

```
@@BASE CSEG    BASE
_intp :
       push    whl             ;Saves register
       push    tde
       push    uup
       push    vvp
       push    ax,bc,rp2,rp3
       movg    whl,sp
       movg    sp,#_buff+10    ;Changes stack pointer
       push    whl
       movw    rp3,#01H        ;Allocates RP3 to variable a Note
       movw    ax,!!_;
       cmpw    ax,rp3
       bne     $L0003
       movg    uup,#_TaskID1
```

Note When the **-QV** option is not specified, the securing/releasing codes of the local variable area are output.

Interrupt Handler for RTOS #pragma rtos_interrupt ...

(Output object of compiler)
When **-ML**, **-QV** is specified (default)

```
        pop     whl                 ;Sets back stack pointer
        movg    sp,whl
        br      !!_ret_wup
L0003 :
        pop     whl                 ;Sets back stack pointer
        movg    sp,whl
        pop     ax,bc,rp2,rp3       ;Restores register
        pop     vvp
        pop     uup
        pop     tde
        pop     whl
        reti

@@VECT06    CSEG    AT    0006H
_@vect06:
        DW      _intp
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the compiler does not support the interrupt handler for RTOS.
- When changing to the interrupt handler for RTOS, modify the program according to the method above.

From this C compiler to another C compiler

- Handled as an ordinary function if the **#pragma rtos_interrupt** specification is deleted.
- When using as an interrupt handler for RTOS, modification is required according to the specification of each compiler.

(27) Interrupt handler qualifier for real-time OS (RTOS)

Interrupt Handler Qualifier for RTOS __rtos_interrupt

FUNCTION

- The function declared with the _ _**rtos_interrupt** qualifier is interpreted as an interrupt handler for RTOS.
- The two RTOS system call calling functions **ret_int/ret_wup** can be called in the function declared with the keywords _ _**rtos_interrupt** (for details of the RTOS system call calling functions, refer to **List of RTOS System Call Calling Functions** described later).
  If the prototype declaration or the entity definition of **ret_int/ret_wup** and **ret_int/ret_wup** are called outside the interrupt handler for RTOS, an error occurs.
- The functions to call the two RTOS system call calling functions **ret_int/ret_wup** are called by an unconditional branch instruction.
- If there is neither **ret_int** nor **ret_wup** in the interrupt handler for RTOS, an error occurs.

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 }
```

[List of the system call calling functions for RTOS]

(a) `void ret_int ( ) ;`

Calls system call **ret_int** for RTOS.

(b)
```
void ret_wup (x) ;
char *x ;
```

Calls system call **ret_wup** for RTOS with x as an argument.

Interrupt Handler Qualifier for RTOS __rtos_interrupt

RESTRICTIONS

callt/callf/noauto/norec/_ _callt/_ _callf/_ _leaf/ _ _interrupt/_ _interrupt_brk/ _ _ pascal/_ _ flash cannot be specified for the interrupt handler for RTOS.

- The RTOS system call calling function names **ret_int/ret_wup** cannot be used for the function names.

CAUTIONS

- Vector addresses cannot be set only by declaring this qualifier.
  The setting of the vector address must be performed separately by 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 this C compiler

- Modification is not required if the compiler does not support interrupt handler for RTOS.
- When changing to interrupt handler for RTOS, modify the program according to the method above.

From this C compiler to another C compiler

- Changes can be made by **#define** (for details, refer to **11.6 Modifications of C Source**). By these changes, interrupt handler qualifiers for RTOS are handled as ordinary variables.
- When using as an interrupt handler for RTOS, modification is required according to the specification of each compiler.

(28) Task function for real-time OS (RTOS)

Task Function for RTOS #pragma rtos_task

FUNCTION

- The function names specified with **#pragma rtos_task** are interpreted as the tasks for RTOS.
- If the function name is specified and the entity definition is not in the same file, an error occurs.
- The preprocessing of the task function for RTOS does not save the registers for frame pointer/register variables. The postprocessing is not output.
- The following RTOS system call calling functions can be used.

[**RTOS system call calling functions]**

(a) `void ext_tsk (void);`

Calls RTOS system call ext_tsk.

However, when **ext_tsk** is called in the **ext_tsk** prototype declaration or entity definition, interrupt function, or interrupt handler for RTOS, an error occurs.

- The RTOS system call calling function of **ext_tsk** is called by an unconditional branch instruction. If **ext_tsk** is issued after the function, the postprocessing is not output.
- When there is no **ext_tsk** in the task function for RTOS and the **-W2** option is specified, a warning message is output.

EFFECT

- The task function for RTOS can be described at the C source level.
- The saving and postprocessing of the register frame pointer/register variable are not output, so the code efficiency is improved.

Task Function for RTOS #pragma rtos_task

USAGE

- Specifies the function name for the following **#pragma** directives.
- The **#pragma** directives are described at the start of the C source.
  However, the following items can be described before the **#pragma** directive.
  - Comments
  - Preprocessing directives that neither define nor reference variables or functions.
- Keywords following **#pragma** can be described either in uppercase or lowercase letters.

```
#pragmaΔrtos_task [Δtask-function-name]
```

RESTRICTIONS

- **callt/callf/noauto/norec/_ _callt/_ _callf/_ _leaf/_ _interrupt/_ _interrupt brk/_ _rtos_interrupt/ _ _ pascal/_ _ flash** cannot be specified for the task function for RTOS.
- The task function for RTOS cannot be called in the same manner as ordinary functions.
  The RTOS system call calling function name **ext_tsk** cannot be used for a function name.
  The task function for RTOS is not supported when the medium model is specified.

EXAMPLE

(C source)

```
#pragma rtos_task func
void main ( ) {
        int a;
        a = 1;
        ext_tsk ();
}
void func ( ) {
        register int r;
        int x;
        x = 1;
        r = 2;
        ext_tsk ();
}
```

Task Function for RTOS #pragma rtos_task

(Output object of compiler)
When **-ML**, **-QV** is specified (default)

```
@@CODE          CSEG
_main :
        push    rp3
        movw    rp3,#01H            ;1
        br      !!_ext_tsk          ;Epilogue is not output.
_func :                             ;Frame pointer is not saved.
        movw    up,#01H             ;1
        movw    rp3,#02H            ;2
        br      !!_ext_tsk          ;Epilogue is not output.
        END
```

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the compiler does not support the task function for RTOS.
- When changing to the task function for RTOS, modify the program according to the method above.

From this C compiler to another C compiler

If the **#pragma rtos_task** specification is deleted, the RTOS task function is used as an ordinary function.
To use as RTOS task function, modification is required according to the specification of each compiler.

(29) Changing function call interface

Changing Function Call Interface -ZO

FUNCTION

- Arguments are passed in accordance with the former function interface specifications (in CC78K4 V1.00 compatible products, only the stack is used). For details of the function interface, refer to **11.7 Function Call Interface**.

USAGE

- The **-ZO** option is specified during compilation.

RESTRICTION

- Modules to which the **-ZO** option is specified and modules to which the **-ZO** option is not specified cannot be linked to one another.

(30) Changing the method of calculating the offset of arrays and pointers

Changing the Method of Calculating the Offset of Arrays and Pointers -QH

FUNCTIONS

- When calculating the offset of arrays and pointers (distance from the start of the array or pointer), if the index is an **int/short** type variable, it is regarded as **unsigned int/unsigned short**, and if the index is a **char** type variable, it is regarded as **unsigned char**.
- Calculates the offset as a positive 64 KB or less.
- However, the ordinary offset calculation is performed if the index is a **long** type variable or a constant.

EFFECT

- The code efficiency is improved by performing unsigned offset calculation.

USAGE

- The **-QH** option is specified during compilation.

RESTRICTIONS

- Access to an object by array elements and pointers can be performed only when the offset is 64 KB or less.
- The offset for the minus direction cannot be calculated.

COMPATIBILITY

From another C compiler to this C compiler

- When the index to arrays and pointers is a **int/short** type variable or **char** type variable and there is access to a minus-direction object or access to an object of more than 64 KB, the index is changed to a **long** type variable. Otherwise, the **-QH** option should not be specified.

From this C compiler to another C compiler

- Modification is not required.

Changing the Method of Calculating the Offset of Arrays and Pointers -QH

EXAMPLE

(C source)

```
int tabi [100];
char tabc [100];
int *iptr;
void main (void) {
         long I = 50;
         int i = 30;
         char c = 2;
         tabi [i] = 1;              /* unsigned offset calculation, 64 KB or less */
        tabc [c] = 2;               /* unsigned offset calculation, 64 KB or less */
        tabi [l] = 3;               /* signed offset calculation */
        *(iptr + i) = 4;            /* unsigned offset calculation, 64 KB or less */
        *(iptr + (-i)) = 5;         /* offset calculation, positive 64 KB or less */
        *(iptr - i) = 6;            /* signed offset calculation */
        *(iptr -10) = 7;            /* signed offset calculation */
        *(iptr + (-10)) = 8;        /* signed offset calculation */
}
```

(Output object of compiler)

When **-ML**, **-QH** is specified (1/3)

```
@@CODE CSEG
_main:
       push     uup
       push     rp3
       push     vvp
; line 6:                long 1 = 50;
       movw     rp3,#032H             ;50
       subw     vp,vp
; line 7:                int i = 30;
       movw     up,#01EH              ;30
; line 8:                char  c= 2;
       mov      c,#02H                ;2
; line 9:
; line 10 :              tabi [i] = 1;      /* unsigned offset calculation, 64 KB or less */
       movw     hl,up
```

Changing the Method of Calculating the Offset of Arrays and Pointers -QH

(Output object of compiler)
When **-ML**, **-QH** is specified (2/3)

```
        Addw  hl,hl             ;Offset calculation only for the lower 2 bytes
        Movw  ax,#01H           ;1
        Movw  _tabi[hl],ax
; line  11 :       tabc [c] = 2;          /* unsigned offset calculation, 64 KB or less */
      mov    a,c
      xch    a,b
      mov    a,c
      mov    _tabc[b],a        ;Offset calculation only for the least significant byte
; line 12 :         tabi [l] = 3;          /* signed offset calculation */
      movw   hl,rp3
      mov    a,r8
      mov    w,a
      addg   whl,whl           ;Offset is 3 bytes, sign is considered
      addg   whl,#_tabi
      movw   ax,#03H           ; 3
      movw   [h],ax
; line 13 :          *(iptr + i) = 4;       /* unsigned offset calculation, 64 KB or less */
      movw   hl,up
      movg   tde,!!_iptr
      addw   hl,hl             ;Offset calculation only for the lower 2 bytes
      addg   tde,whl
      incw   ax
      movw   [de],ax
; line 14 :          *(iptr + (-i)) = 5;    /* offset calculation, positive 64 KB or less */
      subw   ax,ax
      subw   ax,up
      movg   whl,!!_iptr
      movw   de,ax
      mov    t,#00H            ;0
      addw   de,de             ;Offset calculation only for the lower 2 bytes
      addg   whl,tde
      movw   ax,#05H           ;5
      movw   [hl],ax
```

Changing the Method of Calculating the Offset of Arrays and Pointers -QH

(Output object of compiler)
When **-ML**, **-QH** is specified (3/3)

```
; line 15 :              *(iptr - i) = 6 ;          /* signed offset calculation */
        movw    hl,up
        mov     a,h
        cvtbw
        mov     w,a
        movg    tde,!!_iptr
        addg    whl,whl         ; Offset is 3 bytes
        subg    tde,whl
        movw    ax,#06H         ; 6
        movw    [de],ax
; line 16 :              *(iptr - 10) = 7 ;         /* signed offset calculation */
        movg    whl,!!_iptr
        incw    ax
        addg    whl,#0FFFFECH ; -20  ; Offset is a signed constant (–20)
        movw    [hl],ax
; line 17 ;              *(iptr + (-10)) = 8 ;      /* signed offset calculation */
        movg    whl,!!iptr
        incw    ax
        addg    whl,#0FFFFECH ; -20  ; Offset is a signed constant (–20)
        movw    [hl],ax
; line 18 ; }
        pop     vvp
        pop     rp3
        pop     uup
        ret
```

Changing the Method of Calculating the Offset of Arrays and Pointers -QH

(Output object of compiler)

When **-ML**, **-QH** is not specified (1/3)

```
@@CODE CSEG
_main :
        push    uup
        push    rp3
        push    vvp
; line 6:               long I = 50;
        movw    rp3,#032H             ;50
        subw    vp,vp
; line 7:               int i = 30;
        movw    up,#01EH              ;30
; line 8:               char  c= 2;
        mov     c,#02H                ;2
; line 9:
; line 10 :             tabi [i] = 1;       /* unsigned offset calculation, 64 KB or less */
        movw    hl,up
        mov     a,h
        cvtbw
        mov     w,a
        addg    whl,whl
        addg    whl,#_tabi
        movw    ax,#01H               ; 1
        movw    [hl],ax
; line 11 :             tabc [c] = 2;       /* unsigned offset calculation, 64 KB or less */
        mov     a, c
        cvtbw
        movw    hl,ax
        mov     w,a
        addg    whl,#_tabc
        mov     a, c
        mov     [hl],a
; line 12 :             tabi [l] = 3;       /* signed offset calculation */
        movw    hl,rp3
        mov     a,r8
        mov     w,a
        addg    whl,whl
        addg    whl,#_tabi
        movw    ax,#03H               ;3
        movw    [hl],ax
```

Changing the Method of Calculating the Offset of Arrays and Pointers -QH

(Output object of compiler)
When **-ML**, **-QH** is not specified (2/3)

```
; line  13 :          *(iptr + i) = 4;      /* unsigned offset calculation, 64 KB or less */
        movw   hl,up
        movg   tde,!!_iptr
        mov    a,h
        cvtbw
        mov    w,a
        addg   whl,whl
        addg   tde,whl
        movw   ax,#04H       ; 4
        movw   [de],ax
; line  14 :          *(iptr + (-i)) = 5;  /* offset calculation positive 64 KB or less */
        subw   ax,ax
        subw   ax,up
        movg   whl,!!_iptr
        movw   de,ax
        cvtbw
        mov    t,a
        addg   tde,tde
        addg   whl,tde
        movw   ax,#05H       ; 5
        movw   [hl],ax
; line  15 :          *(iptr - i) = 6;      /* signed offset calculation */
        movw   hl,up
        mov    a,h
        cvtbw
        mov    w,a
        movg   tde,!!_iptr
        addg   whl,whl
        subg   tde,whl
        movw   ax,#06H       ; 6
        movw   [de],ax
```

Changing the Method of Calculating the Offset of Arrays and Pointers -QH

(Output object of compiler)
When **-ML**, **-QH** is not specified (3/3)

```
; line 16 :              *(iptr - 10) = 7;          /* signed offset calculation */
        movg    whl,!!_iptr
        incw    ax
        addg    whl,#0FFFFECH           ;-20
        movw    [hl],ax
; line 17 :    *(iptr + (-10)) = 8;               /* signed offset calculation */
        movg    whl,!!_iptr
        incw    ax
        addg    whl,#0FFFFECH           ;-20
        movw    [hl],ax
; line   18 : }
        pop     vvp
        pop     rp3
        pop     uup
        ret
```

COMPATIBILITY

From another C compiler to this C compiler

- When the index to arrays and pointers is a **int/short** type variable or **char** type variable and there is access to a minus-direction object or access to an object of more than 64 KB, the index is changed to a **long** type variable. Otherwise, the **-QH** option should not be specified.

From this C compiler to another C compiler

- Modification is not required.

(31) Pascal function

Pascal Function _ _pascal

FUNCTION

- Generates the code that corrects the stack used for placing of arguments when a function is called on the called function side, not on the side calling the function.

EFFECT

- Object code can be shortened if a lot of function calls appear.

USAGE

- When a function is declared, a _ _**pascal** attribute is added to the beginning.

RESTRICTIONS

- The pascal function does not support variable length arguments. If a variable length argument is defined, a warning is output and the _ _**pascal** keyword is disregarded.
- In a pascal function, the keywords **norec/_ _interrupt/_ _interrupt_brk/_ _rtos_interrupt/_ _flash** cannot be specified. If they are specified, in the case of the **norec** keyword, the _ _**pascal key** word is disregarded and in the case of the _ _**interrupt/_ _interrupt_brk/_ _rtos_interrupt/_ _flash** keywords, an error is output.
- The old specification function interface specification option (**-ZO**) does not support the pascal function. When pascal functions are used, if **-ZO** is specified, a warning message is output at the first place where a _ _**pascal** key word appears and the _ _**pascal** keywords in the input file are disregarded.
- If a prototype declaration is incomplete, it won't operate normally, so a warning message is output when a pascal function's physical definition or prototype declaration is missing.

EXPLANATION

- The **-ZR** option enables the change of all functions to the pascal function. However, if the pascal function is used to change functions that have few function calls, object code may increase.

EXAMPLE

(C source)

```
      _ _pascal int func(int a, int b, int c);
      void main()
      {
              int ret_val;

  ret_val = func(5, 10, 15);
```

Pascal Function _ _pascal

(C source) (continued)

```
}
_ _pascal int func(int a, int b, int c)
{
        return (a + b + c);
}
```

(Output object of compiler)
With large model

```
_main:
            push    rp3
            movw    ax,#0FH ;
            push    ax      ;
            mov     x,#0AH  ;
            push    ax      ;
            mov     x,#05H  ; With the argument, a 4-byte stack is consumed.
            call    $!_func
                            ; Here stack correction is not performed.
            movw    rp3,bc
            pop     rp3
            ret

_func:
            push    rp3
            movw    rp3,ax
            movw    ax,[sp+5]
            addw    ax,rp3
            movw    bc,ax
            movw    ax,[sp+7]
            addw    bc,ax
            pop     rp3
            pop     whl     ; Obtain the return address.
            pop     ax,rp2  ; The 4-byte stack consumed on the calling side is corrected.
            br      whl     ; Branch to the return address.
```

Pascal Function _ _pascal

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the reserved word _ _ **pascal** is not used.
- When changing to the pascal function, modify the program according to the method above.

From this C compiler to another C compiler

- Compatibility is maintained by using **#define**.
- By this conversion, the pascal function is regarded as an ordinary function.

(32) Automatic pascal functionization of the function call interface

Automatic Pascal Functionization of the Function Call Interface -ZR

FUNCTION

- With the exception of **norec/_ _interrupt/_ _interrupt_brk/_ _rtos_interrupt/_ _flash** and functions with variable length arguments, _ _**pascal** attributes are added to all functions.

USAGE

- The **-ZR** option is specified during compilation.

RESTRICTIONS

- The old specification function interface specification option (**-ZO**) cannot be used at the same time. If this is used, a warning message is output and the **-ZR** option is ignored.
- Modules in which the **-ZR** option is specified and modules in which the **-ZR** option is not specified cannot be linked. If a link is executed, it results in a link error.

Remark For details of the pascal function call interface, refer to **11.7.5 Pascal function call interface**.

(33) Flash area allocation method

Flash Area Allocation Method -ZF

Caution Do not use this flash function for devices that have no flash area self-rewrite function. Operation is not guaranteed if it is used.
This function enables the flash memory rewrite function of devices.

FUNCTIONS

- Generates an object file located in the flash area.
- External variables in the flash area cannot be referenced from the boot area.
- External variables in the boot area can be referenced 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

- The **-ZF** option is specified during compilation.

RESTRICTION

Use startup routines or library for the flash area.

(34) Flash area branch table

Flash Area Branch Table #pragma ext_table

Caution Do not use this flash function for devices that have no flash area self-rewrite function. Operation is not guaranteed if it is used. This function enables the flash memory rewrite 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.
- The branch instruction, which is one of the branch table elements, occupies 4 bytes of area. 32 from the first address of the branch table are reserved as dedicated interrupt functions. Ordinary functions are located after the "first address of branch table +4 * 32."
- The branch table occupies 4* (32 + **ext_func** ID max. value + 1) bytes of area. For the **ext_func** ID value, refer to **11.5 (35) Function call function from the boot area to the flash area**.

EFFECT

- A startup routine and interrupt function can be located in the flash area.
- A function call can be performed from the boot area to the flash area.

USAGE

- The following **#pragma** directive specifies the first address of the flash area branch table.

```
#pragma Δext_table Δ branch-table-first-address
```

Describe the **#pragma** directive at the beginning of the C source.

- The following items can be described before the **#pragma** directive.
  - Comments
  - **#pragma** directive other than **#pragma ext_func, #pragma vect** with **–ZF** specification, **#pragma interrupt**, or **#pragma rtos_interrupt**.
  - Directives not to generate the definition/reference of variables or functions among the preprocessing directives.

Flash Area Branch Table #pragma ext_table

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 occurs.
- The first address of the branch table is assumed to be 0x80 to 0xff80. However, match the first address value with the flash start address which is specified by the **-ZB** linker option. If the address does not match, it results in a link error.
- It is necessary to reconfigure the library for interrupt vectors (**_@vect100** to **_@vect3e**) in accordance with the specified first address of the branch table. The default is 4000H in the interrupt vector library. To specify a value other than 0x4000, reconfigure the library as shown below.

1. Change the place of H in ITBLTOP EQU 4000H of **vect.inc** in the \NECTools32\SRC\CC78K4\SRC directory to the specified address.
2. Run \NECTools32\SRC\CC78K4\BAT/repvect.bat in DOS prompt, and update library using the assembler, etc. Copy the updated library \NECTools32\SRC\CC78K4\LIB to \NECTools32\LIB78K4 to be used for linking.

Caution The above directory may differ depending on the installation method.

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if **#pragma ext_table** is not used.
- When specifying the first address of the flash area branch table, change the address according to the method above.

From this C compiler to another C compiler

- Delete the **#pragma ext_table** instruction or delimit it with **#ifdef**.
- When specifying the first address of the flash area branch table, the following modification is required.

Flash Area Branch Table #pragma ext_table

EXAMPLE

To generate the branch table after the address 4000H and allocate the interrupt function.

(C source)

```
  #pragma ext_table 0x4000
  #pragma interrupt INTP0 intp

void intp()
{
}
```

(Output object of compiler)

(a) To allocate the interrupt function to the boot area (no **-ZF** specification).

```
            PUBLIC  _@vect06
            PUBLIC  _intp

@@BASE      CSEG    BASE
_intp:
                    reti

@@VECT06    CSEG    AT   0006H
_@vect06:
            DW              _intp
```

- Set the first address of the interrupt function in the interrupt vector table.

Flash Area Branch Table #pragma ext_table

(b) To allocate the interrupt vector table to the flash area (**-ZF** specified).

```
          PUBLIC  _intp

@ECODE    CSEG
_intp:
          reti

@EVECT06          CSEG      AT    0400CH
          br      !!_intp
```

(Library for interrupt vector 06)

```
          PUBLIC  _@vect06

@@VECT06 CSEG     AT    0006H
_vect06:
          DW      400CH
```

- Set the first address of the interrupt function in the branch table.
- The first address of the branch table is 4000H and the interrupt vector address (2 bytes) is 0006H, so the address of the branch table becomes 4000H + 4*(0006H/2).
- Setting the 400CH address in the interrupt vector table is performed by the interrupt vector library.

(35) Function call function from the boot area to the flash area

Function Call Function from the Boot Area to the Flash Area #pragma ext_func

Caution Do not use this flash function for devices that have no flash area self-rewrite function. Operation is not guaranteed if it is used. This function enables the flash memory rewrite function of devices.

FUNCTIONS

- Function calls from the boot area to the flash area are executed via the flash area branch table.
- Functions in the boot area can be called directly from the flash area.

EFFECT

- It becomes possible to call a function in the flash area from the boot area.

USAGE

- The following **#pragma** directive specifies the function name and ID value in the flash area called from the boot area.

```
#pragma Δ ext_func Δ function-name Δ ID value
```

This **#pragma** directive is described at the beginning of the C source. The following items can be described before this **#pragma** directive.

- Comments
- Directives that do not generate the definition/reference of variables or functions among the preprocessing directives.

RESTRICTIONS

- The ID value is set to 0 to 255 (0xFF).
- If **#pragma ext_table** does not exist before **#pragma ext_func**, it results in an error.
- If the same function has a different ID value or a different function has the same ID value, an error occurs. (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** and **callf** functions can only be located in the boot area. If the **callt** and **callf** functions are defined in the flash area (when the **-ZF** option is specified), it results in an error.

Function Call Function from the Boot Area to the Flash Area #pragma ext_func

COMPATIBLITY

From another C compiler to this C compiler

- Modification is not required if the **#pragma ext_func** is not used.
- When performing the function call from the boot area to the flash area, modify the program according to the method above.

From this C compiler to another C compiler

- Delete the **#pragma ext_func** instruction or delimit it with **#ifdef**.
- When performing the function call from the boot area to the flash area, the following modification is required.

EXAMPLE

In the case that the branch table is generated after address 4000H and functions f1 and f2 in the flash area are called from the boot area.

(C source)

(1) Boot area side

```
#pragma ext_table 0x4000
#pragma ext_func f1 3
#pragma ext_func f2 4

extern void f1(void);
extern void f2(void);

void func()
{
 f1();
 f2();
}
```

Function Call Function from the Boot Area to the Flash Area #pragma ext_func

(2) Flash area side

```
#pragma ext_table 0x4000
#pragma ext_func f1 3
#pragma ext_func f2 4

void f1()
{
}
void f2()
{
}
```

- #pragma ext_func f1 3 means that the branch destination to function f1 is located in branch table address 4000H + 4*32 + 4*3.
- #pragma ext_func f2 4 means that the branch destination to function f2 is located in branch table address 4000H + 4*32 + 4*4.
- 4*32 bytes from the beginning of the branch table is exclusively for interrupt functions (including the startup routine).

(Output object of compiler)

(1) Boot area side (without **-ZF** specification)

```
@@CODE          CSEG
_func:
          call  !0408CH
          call  !04090H
          ret
```

(2) Flash area side (with **-ZF** specification)

```
@ECODE          CSEG
_f1:
          ret
_f2:
          ret

@EXT03          CSEG    AT    0408CH
          br    !!_f1
          br    !!_f2
```

(36) Firmware ROM function

Firmware ROM Function _ _flash

Caution Do not use this flash function for devices that have no flash area self-rewrite function. Operation is not guaranteed if it is used. This function enables the flash memory rewrite function of devices.

FUNCTIONS

- This calls a firmware ROM function that 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 via the register and the second and subsequent arguments are transferred to the stack. The first argument's register is as follows.

1, 2-byte integer	AX
3-byte integer	WHL
4-byte integer	AX (lower integer), RP2 (higher integer)

- The size of the pointer passed to the stack after the second argument is three bytes.

EFFECT

- Operations related to the firmware ROM function can be described at the C source level.

USAGE

- _ _**flash** attributes are added to the top during an interface library prototype declaration.

RESTRICTIONS

- Function calls by a function pointer are not supported.
- When the old specification function interface specification option (**-ZO**) is specified, it results in an error.
- When a function with _ _**flash** is defined, it results in an error.

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the reserved word _ _**flash** is not used.
- When changing the firmware ROM function, modify the program according to the method above.

From this C compiler 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.

(37) Method of int expansion limitation of argument/return value

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 ()
{
        c = func1 (d, e);
        c = func2 (d, e);
}
unsigned char func1 (unsigned char x, unsigned char y)
{
        return x + y;
}
```

Method of int Expansion Limitation of Argument/Return Value -ZB

(Output object of compiler)

When **-ZB** is specified

```
_main:
; line  5:  c = func1 (d, e);
     mov    x, !!_e                 ;Do not execute int expansion
     push   ax                      ;Do not execute int expansion
     mov    x, !!_d
     call   $!_func1
     pop    ax
     mov    !!_c,c
; line  6   c = func2 (d, e);
     mov    x, !!_e
     mov    a, #00H ; 0             ;Execute int expansion since there is no prototype declaration
     push   ax
     mov    x, !!_d
     call   !!_func2
     pop    ax
     mov    !!_c,c
; line  7:  }
     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 this C compiler

- If the prototype declarations for all definitions of function bodies are not correctly performed, perform correct prototype declaration. Alternatively, do not specify the **-ZB** option.

From this C compiler to another C compiler

- No modification is required.

(38) Memory manipulation function

Memory Manipulation Function #pragma inline

FUNCTION

- An object file is generated by the output of the standard library memory manipulation functions **memcpy**, **memset, memchr, and memcmp** with direct inline expansion instead of function call.
- When there is no **#pragma** directive, the code that calls the standard library functions is generated.

EFFECT

- Compared with when a standard library function is called, the execution speed is improved.
- Object code is reduced if a constant is specified for the specified character number.

USAGE

- The function is described in the source in the same format as a function call.
- The following items can be described before **#pragma inline**.
  - Comments
  - Other **#pragma** directives
  - Preprocessing directives that do not generate variable definitions/references or function definitions/references

EXAMPLE

(C source)

```
#pragma inline
char ary1[100], ary2[100];
void main()
{
     memset(ary1, 'A', 50);
     memcpy(ary1, ary2, 50);
}
```

Memory Manipulation Function #pragma inline

(Output object of compiler)

When **-MS** is specified

```
_main:
; line     7 :  memset(ary1, 'A', 50);
        movw    de,#_ary1
        mov     c,#032H ; 50
        mov     a,#041H ; 65
        mov     [de+],a
        dbnz    c,$$-1
; line     8 :  memcpy(ary1, ary2, 50);
        movw    de,#_ary1
        mov     c,#032H ; 50
        movw    hl,#_ary2
        mov     a,[hl+]
        mov     [de+],a
        dbnz    c,$$-2
; line     9 :
; line    10 :  p = memchr(ary1, 'B', 50);
        mov     c,#032H ; 50
        movw    de,#_ary1
        mov     a,#042H ; 66
        cmp     a,[de]
        bz      $L0006
        incw    de
        dbnz    c,$$-5
        subw    de,de
L0006:
        movw    !_p,de
; line    11 :  i = memcmp(ary1, ary2, 100);
        mov     c,#064H ; 100
        movw    de,#_ary1
        movw    hl,#_ary2
        mov     a,[de+]
        sub     a,[hl+]
        bnz     $L0008
        dbnz    c,$$-5
L0008:
        subc    x,x
        xch     a,x
        movw    !_i,ax
; line    12 : }
      ret
```

Memory Manipulation Function #pragma inline

(Output object of compiler)

When **-MM** is specified

```
_main:
; line     7 :  memset(ary1, 'A', 50);
        movw    de,#LOWW _ary1
        mov     c,#032H ; 50
        mov     a,#041H ; 65
        mov     [de+],a
        dbnz    c,$$-1
; line     8 :  memcpy(ary1, ary2, 50);
        movw    de,#LOWW _ary1
        mov     c,#032H ; 50
        movw    hl,#LOWW _ary2
        mov     w,#0FH  ; 15
        mov     a,[hl+]
        mov     [de+],a
        dbnz    c,$$-2
; line     9 :
; line    10 :  p = memchr(ary1, 'B', 50);
        mov     c,#032H ; 50
        movw    de,#LOWW _ary1
        mov     a,#042H ; 66
        cmp     a,[de]
        bz      $L0006
        incw    de
        dbnz    c,$$-5
        subw    de,de
L0006:
        movw    !!_p,de
; line    11 :  i = memcmp(ary1, ary2, 100);
        mov     c,#064H ; 100
        movw    de,#LOWW _ary1
        movw    hl,#LOWW _ary2
        mov     w,#0FH  ; 15
        mov     a,[de+]
        sub     a,[hl+]
        bnz     $L0008
        dbnz    c,$$-5
L0008:
        subc    x,x
        xch     a,x
        movw    !!_i,ax
; line    12 : }
      ret
```

Memory Manipulation Function #pragma inline

(Output object of compiler)

When **-ML** is specified

```
_main:
; line     7 :  memset(ary1, 'A', 50);
        movg    tde,#_ary1
        mov     c,#032H ; 50
        mov     a,#041H ; 65
        mov     [de+],a
        dbnz    c,$$-1
; line     8 :  memcpy(ary1, ary2, 50);
        movg    tde,#_ary1
        mov     c,#032H ; 50
        movg    whl,#_ary2
        mov     a,[hl+]
        mov     [de+],a
        dbnz    c,$$-2
; line     9 :
; line    10 :  p = memchr(ary1, 'B', 50);
        mov     c,#032H ; 50
        movg    tde,#_ary1
        mov     a,#042H ; 66
        cmp     a,[de]
        bz      $L0006
        incg    tde
        dbnz    c,$$-6
        subg    tde,tde
L0006:
        movg    !!_p,tde
; line    11 :  i = memcmp(ary1, ary2, 100);
        mov     c,#064H ; 100
        movg    tde,#_ary1
        movg    whl,#_ary2
        mov     a,[de+]
        sub     a,[hl+]
        bnz     $L0008
        dbnz    c,$$-5
L0008:
        subc    x,x
        xch     a,x
        movw    !!_i,ax
; line    12 : }
            ret
```

Memory Manipulation Function #pragma inline

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the memory manipulation function is not used.
- When changing the memory manipulation function, modify the program according to the method above.

From this C compiler to another C compiler

- Delete the **#pragma inline** directive or delimit it with **#ifdef**.

(39) callf two-step branch function

callf Two-Step Branch Function -ZG

FUNCTION

- A function body to which the **callf/_ _callf** attribute is added is not allocated to the **callf** area from 800H to 0FFFH, a branch instruction to the function body is allocated to the **callf** area, and the code to call the branch instruction using the **callf** instruction is generated.

EFFECT

- Compared to the case when allocating a function body to the **callf** area, the **callf/_ _callf** attribute can be added to many more functions. Therefore, this function can shorten the object code if many functions that include call functions are frequently used.

USAGE

- The **-ZG** option is specified during compilation.

RESTRICTIONS

- Modules in which the **-ZG** option is specified and modules in which the **-ZG** option is not specified cannot be linked.
- The two-step branch table consumes 4 bytes per function when the **-MM/ML** option is specified, and 3 bytes when the **-MS** option is specified. The maximum number of **callf** functions that can be allocated when the **-ZG** option is specified per load module and the total number of **callf** functions per linked module are as follows.

- When the **-MM/ML** option is specified: 512
- When the **-MS** option is specified: 682

EXAMPLE

(C source 1)

```
_ _callf extern int fsub();
void main()
{
        int ret_val;
        ret_val = fsub();
}
```

callf Two-Step Branch Function -ZG

(C source 2)

```
_ _callf int fsub()
   {
        int val = 1;
    return val;
}
```

(Output object of compiler)

With large or medium model

(C source 1)

```
        EXTRN   ?fsub   ; Declaration

        callf   !?fsub  ; Call
```

(C source 2)

```
        PUBLIC  _fsub   ; Declaration
        PUBLIC  ?fsub   ; Declaration

@@CALF  CSEG    FIXED
?fsub:  br      !!_fsub ; Branch table

@@CODE  CSEG
_fsub:                  ; Function definition
     .
     .
 Function body
     .
     .
```

callf Two-Step Branch Function -ZG

(Output object of compiler)

With small model

(C source 1)

```
        EXTRN   ?fsub    ; Declaration

        Callf   !?fsub   ; Call
```

(C source 2)

```
        PUBLIC  _fsub    ; Declaration
        PUBLIC  ?fsub    ; Declaration

@@CALFS CSEG    FIXEDA
?fsub:  br      !_fsub   ; Branch table

@@CODES CSEG    BASE     ; Function definition
_fsub:
          .
          .
     Function body
          .
          .
```

(40) Automatic callf functionization of function call interface

Automatic Callf Functionization of Function Call Interface -ZH

FUNCTION

- The _ _**callf** attribute is added to all functions except for the **callt/_ _callt/_ _interrupt/_ _interrupt_brk/_ _rtos_interrupt** functions.

USAGE

- The **-ZH** option is specified during compilation.

RESTRICTIONS

- The **-ZF** option for the flash area allocation specification cannot be specified at the same time.
  If specified, a warning message is output and the **-ZH** option is ignored.
- The standard library that supports the **-ZF** option is not available. Sources that include the standard library cannot be linked using the **-ZF** option during compilation.

(41) Three-byte address reference/generation function

Three-Byte Address Reference/Generation Function #pragma addraccess

FUNCTION

- A code that references the highest byte and the lower 2 bytes of a 3-byte address, and a code that generates a 3-byte address from the value of the highest byte and the lower 2 bytes are output to an object directly with inline expansion and an object file is created.
- If the **#pragma** directive is not added, the three-byte address reference/generation function is regarded as an ordinary function.

EFFECT

- Three-byte address reference/generation can be performed with a short code without using a complex cast description.

USAGE

- Describe the **#pragma addraccess** directive at the beginning of the C source.
- Describe the **#pragma addraccess** directive in the C source in the same manner as a function call.
- The following items can be described before the **#pragma addraccess** directive.
  (1) Comments
  (2) Other **#pragma** directives
  (3) Among the preprocessing directives, those that do not generate a variable definition/reference or function definition/reference.
- The keywords following **#pragma addraccess** can be described in either uppercase or lowercase letters.

The following three names can be used for the three-byte address reference/generation function name.

- `FP_SEG`
- `FP_OFF`
- `MK_FP`

[List of function names for three-byte address reference/generation]

(1) unsigned char FP_SEG(void *addr);
The value of the most significant byte of a three-byte address pointed by addr is obtained.
(2) unsigned int FP_OFF(void *addr);
The values of the lower 2 bytes of a three-byte address pointed by addr are obtained.
(3) void *MK_FP(unsigned char seg, unsigned int offset);
The address value of the three-byte address having the value pointed by seg as the most significant byte, and the value pointed by offset as the lower 2 bytes.

Three-Byte Address Reference/Generation Function #pragma addraccess

RESTRICTIONS

- The function names for three-byte address reference/generation cannot be used as the function names.
- Describe the three-byte address reference/generation function in uppercase letters. If lowercase letters are used, it is regarded as an ordinary function.
- When the small or medium model is specified, **#pragma addraccess** is ignored and the three-byte address reference/generation function is not supported.

EXAMPLE

```
#pragma addraccess
unsigned char seg;
unsigned int  offset;
unsigned char ary[10];
unsigned char *p;
void main()
{
       seg    = FP_SEG(ary);          /* Most significant byte value */
       offset = FP_OFF(ary);          /* Value of lower 2 bytes */

       p = MK_FP(seg, offset);        /* Generates 3-byte address */
}
```

(Output object of compiler)

```
@@CODE  CSEG
_main:
; line     8 :  seg    = FP_SEG(ary);        /* Most significant byte value */
        mov     a,#HIGHW _ary
        mov     !!_seg,a
; line     9 :  offset = FP_OFF(ary);        /* Value of lower 2 bytes */
        movw    ax,#LOWW _ary
        movw    !!_offset,ax
; line    10 :
; line    11 :  p = MK_FP(seg, offset);      /* Generates 3-byte address */
        mov     a,!!_seg
        mov     w,a
        movw    hl,!!_offset
        movg    !!_p,whl
; line    12 : }
        ret
```

Three-Byte Address Reference/Generation Function #pragma addraccess

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the three-byte address reference/generation function is not used.
- When specifying the three-byte address reference/generation function, modify the function according to the method above.

From this C compiler to another C compiler

- Delete the **#pragma addraccess** statement or delimit it with **#ifdef**.
  The three-byte address reference/generation function name can be used as the function name.
- When specifying the three-byte address reference/generation function, modify the function conforming to the specification of the C compiler.

(42) Absolute address allocation specification

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.
- The _ _**directmap** variable in the C source is treated as an ordinary 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 below.

With small model

Address Specification Range	Secured Area Range	Duplication Check Range
0x80 to 0xFFFF	0xFD00 to 0xFEFF	0xF000 to 0xFEFF

With large model (**-CS0** specified)

Address Specification Range	Secured Area Range	Duplication Check Range
0x80 to 0xFFFFFF	0xFD00 to 0xFEFF	0xF000 to 0xFEFF

With large model (**-CS15** specified)

Address Specification Range	Secured Area Range	Duplication Check Range
0x80 to 0xFFFFFF	0xFFD00 to 0xFFEFF	0xFF000 to 0xFFEFF

With medium model (**-CS0** specified)

Address Specification Range	Secured Area Range	Duplication Check Range
0xF000 to 0xFFFF	0xFD00 to 0xFEFF	0xF000 to 0xFEFF

With medium model (**-CS15** specified)

Address Specification Range	Secured Area Range	Duplication Check Range
0xFF000 to 0xFFFFF	0xFFD00 to 0xFFEFF	0xFF000 to 0xFFEFF

- If the address specification is outside the address specification range, an **F799** error is output.
- If the allocation address of a variable declared by _ _**directmap** is duplicated and is within the duplication check range, a **W762** warning message is output and the name of the duplicated variable is displayed.
- If the address specification range is inside the **saddr1** area, the _ _**sreg1** declaration is made automatically and the **saddr1** instruction is generated. If the address specification range is inside the **saddr2** area, the _ _**sreg** declaration is made automatically and the **saddr2** instruction is generated.
- When the **-CSA** option is specified, a **W338** warning message is output and the _ _**directmap** declaration in the file is ignored.

EFFECT

One or more variables can be allocated to the same arbitrary address.

Absolute Address Allocation Specification _ _directmap

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;
_ _directmap _ _sreg1  Type name    Variable name                     = Allocation address specification;
_ _directmap _ _sreg1  static       Type name    Variable name        = Allocation address specification;
```

- If _ _**directmap** is declared for a structure/union/array, specify the address in braces {}.
- _ _**directmap** does not have to be declared in a module in which a _ _**directmap** external variable is referenced, so only declare **extern**.

```
extern  Type name Variable name;
extern  _ _sreg   Type name    Variable name;
extern  _ _sreg1  Type name    Variable name;
```

- To generate the **saddr2** instruction in a module in which a _ _**directmap** external variable allocated inside the **saddr2** area is referenced, _ _**sreg** must be used together to make **extern**_ _**sreg** Type name Variable name;.
- To generate the **saddr1** instruction in a module in which a _ _**directmap** external variable allocated inside the **saddr1** area is referenced, _ _**sreg1** must be used together to make **extern**_ _**sreg1** Type name Variable name;.

EXAMPLE

(C source)

```
_ _directmap char c = 0xff000;
_ _directmap _ _sreg char d = 0xffd20;
_ _directmap _ _sreg char e = 0xffd21;
_ _directmap struct x
        char a;
        char b;
        char c;
} xx = {0xffe30};
void main()
{
        c = 1;
        d = 0x12;
        e.5 = 1;
        xx.a = 5;
        xx.c = 10;
}
```

Absolute Address Allocation Specification _ _directmap

(Output object)

```
          PUBLIC _c
          PUBLIC _d
          PUBLIC  _e
          PUBLIC _xx
          PUBLIC _main
_c        EQU    0FF000H            ; Addresses for variables declared by _ _directmap
_d        EQU    0FFD20H            ; are defined by EQU
_e        EQU    0FFD21H            ;
_xx       EQU    0FFE30H            ;
          EXTRN  _ _mffd20          ; EXTRN output for linking secured area modules
          EXTRN  _ _mffd21          ;
          EXTRN  _ _mffe30          ;
          EXTRN  _ _mffe31          ;
          EXTRN  _ _mffe32          ;

@@CODE    CSEG
_main:
; line    11 :  c = 1               ;
          mov     !_c,#01H          ; 1
; line    12 :  d = 0x12            ;
           mov    _d,#012H          ; saddr2 instruction output because address
; line    13 :  e.5 = 1             ; specified in saddr2 area
          set1   _e.5               ; Bit manipulation possible because _ _sreg also used
; line    14 :  xx.a = 5            ;
          mov    _xx,#05H           ; saddr1 instruction output because address specified
; line    15 :  xx.c = 10           ; in saddr1 area
          mov    _xx+2,#0AH         ; saddr1 instruction output because address specified
                                    ; in saddr1 area
; line    16 : }
        ret
```

Absolute Address Allocation Specification _ _directmap

RESTRICTIONS

- _ _**directmap** cannot be specified for function arguments, return values, or automatic variables. If it is specified in these cases, an error occurs.
- 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.

COMPATIBILITY

From another C compiler to this C compiler

- Modification is not required if the keyword _ _**directmap** is not used.
- When changing to the _ _**directmap** variable, modify the program according to the method above.

From this C compiler to another C compiler

- Compatibility can be attained using **#define** (refer to **11.6 Modifications of C Source** for details).
- When _ _**directmap** is being used as the absolute address allocation specification, modify the program according to the specifications of each compiler.

11.6 Modifications of C Source

By using the extended functions of this C compiler, efficient object generation can be realized. However, these extended functions are intended to cope with the 78K/IV Series. 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 this C compiler and vice versa is explained.

From another C compiler to this C compiler

- **#pragma**[Note]
  If the other C compiler supports the **#pragma** preprocessing 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 **#pragma** is identified as a directive to the compiler. If the compiler does not support this directive, the **#pragma** directive is ignored and compilation will continue until it properly ends.

From this C compiler to another C compiler

Because this C compiler has added keywords as the extended functions, the C source must be made portable to the other C compiler by deleting such keywords or delimiting them with **#ifdef**.

EXAMPLE

<1> To invalidate a keyword (same applies to **callf**, **sreg**, **noauto**, and **norec** etc.)

```
#ifndef     _ _K4_ _
            #define callt       /* makes callt an ordinary function */
#endif
```

<2> To change from one type to another

```
#ifndef     _ _K4_ _
            #define bit char    /* changes bit type to char type variable */
#endif
```

11.7 Function Call Interface

The following items will be explained concerning the interface between functions when a function is called.

1. Return value (common in all the functions)

2. Ordinary function call interface
   - Passing arguments
   - Location and order of storing arguments
   - Location and order of storing automatic variables

3. **noauto** function call interface
   - Passing arguments
   - Location and order of storing arguments
   - Location and order of storing automatic variables

4. **norec** function call interface
   - Passing arguments
   - Location and order of storing arguments
   - Location and order of storing automatic variables

5. Pascal function call interface

11.7.1 Return value

The function called stores the return value in the registers and carry flags as shown in Table 11-27.

Table 11-27. Storage Location of Return Values

Type \ Model	Small Model	Medium Model	Large Model
1-byte integer 2-byte integer	BC	BC	BC
4-byte integer	BC (Lower) RP2 (Higher)	BC (Lower) RP2 (Higher)	BC (Lower) RP2 (Higher)
Pointer	BC	BC (data pointer) WHL (function pointer)	TDE
Structure, union	BC (structure copied to the area specific to the function, the start address of the union)	BC (structure copied to the area specific to the function, the start address of the union)	TDE (structure copied to the area specific to the function, the start address of the union)
1 bit	CY (carry flag)	CY (carry flag)	CY (carry flag)
Floating-point number (**float** type)	BC (Lower) RP2 (Higher)	BC (Lower) RP2 (Higher)	BC (Lower) RP2 (Higher)
Floating-point number (**double** type)	BC (Lower) RP2 (Higher)	BC (Lower) RP2 (Higher)	BC (Lower) RP2 (Higher)

11.7.2 Ordinary function call interface

When all the arguments are allocated to registers and there is no automatic variable, the ordinary function call interface is the same as **noauto** function call interface.

(1) Passing arguments

(a) When the -ZO option is not specified (default)

- On the function call side, both the arguments declared with registers and the ordinary arguments are passed in the same manner. The second and subsequent arguments are passed via a stack, and the first argument is passed via a register or stack.
- The location where the first argument is passed is shown in **Table 11-28**.

Table 11-28. Location Where First Argument Is Passed (On Function Call Side)

Type \ Option	When -ZO Is Not Specified	When -ZO Is Specified
1-byte integer[Note] 2-byte integer	AX	Passed via a stack
3-byte integer	WHL Small model is passed via a stack	Passed via a stack
4-byte integer[Note]	AX, RP2	Passed via a stack
Floating-point number (**float** type)	AX, RP2	Passed via a stack
Floating-point number (**double** type)	AX, RP2	Passed via a stack
Other	Passed via a stack	Passed via a stack

Note 1- to 4-byte data includes structures, unions, and pointers.

(b) When the -ZO option is specified

- On the function call side, arguments declared with a register are passed via a register, and ordinary arguments are passed via a stack. For the registers used for passing, refer to **Table 11-30**.

(2) Location and order of storing arguments

- There are two types of arguments: arguments allocated to registers and ordinary arguments. Arguments allocated to registers are the arguments declared with registers and the arguments when **-QV** is specified.
- The arguments not allocated to registers are allocated to stacks. The arguments allocated to stacks are placed on the stack sequentially from the last argument.

(a) When the -ZO option is not specified

- Saving and restoring registers to which arguments are allocated is performed on the function definition side.
- When **-QV** option is specified, the ordinary arguments are also allocated to registers regarding they are declared with registers.
- The ordinary arguments are allocated to a stack. When the arguments are passed via stacks, the area where the arguments are passed (stack) is used as the area to which arguments are allocated.
- On the function definition side, the arguments that are passed via a register or stack are stored in the area to which arguments are allocated.
- Arguments with more references together with register variables are allocated to registers. When the **-QF** and **-ML** options are specified, however, a second or subsequent argument whose size is less than 4-bytes and number of references is two or less is not always allocated to a register.

Table 11-29. List of Storing Arguments (On Function Definition Side, When -ZO Is Not Specified)

Option \ Model	Small Model, Medium Model[Note]	Large Model
When **-QF** is specified	RP3, VP, UP	RP3, VVP, UUP
When **-QF** is not specified	RP3, VP	RP3, VVP

Note With the medium model, the function pointer (3 bytes) cannot be used as a register argument.

(Order of allocation)

- With small model, medium model, when **-QF** is specified
  - **char**, **int**, **short**, **enum** type: If there is **long**, **float**, **double** type argument, in the order of UP, RP3, VP
  - **char, int, short,** enum type: If there is no **long**, **float**, **double** type argument, in the order of RP3, UP, VP
  - Pointer type: In the order of UP, VP, RP3
  - **long**, **float**, **double** type: RP3 (lower), VP (higher)
- With small model, medium model, when **-QF** is not specified
  - **char**, **int**, **short**, **enum** type: In the order of RP3, VP
  - Pointer type: In the order of VP, RP3
  - **long**, **float**, **double** type: RP3 (lower), VP (higher)
- With large model, when **-QF** is specified
  - **char**, **int**, **short**, **enum** type: If there is **long**, **float**, **double** type argument, in the order of UP, RP3, VP
  - **char, int, short, enum** type: If there is no **long**, **float**, **double** type argument, in the order of RP3, UP, VP
  - Pointer type: In the order of UUP, VVP
  - **long**, **float**, **double** type: RP3 (lower), VP (higher)

- With large model, when **-QF** is not specified
  char, **int**, **short**, **enum** type: In the order of RP3, VP
  Pointer type: In the order of VVP
  long, **float**, **double** type: RP3 (lower), VP (higher)

(b) When the -ZO option is specified

- The locations where arguments are passed on the function call side and the function definition side are the location where arguments are allocated.
- As long as there are allocable registers, the arguments declared with registers are allocated to registers.
- The saving and restoring of registers to which arguments are allocated is performed before and after the function call.

Table 11-30. List of Storing Arguments (On Function Definition Side, When -ZO Is Specified)

Option \ Model	Small Model	Large Model
When **-QF** is specified	RP3, VP, UP	RP3, VVP
When **-QF** is not specified	RP3, VP	RP3, VVP

(Order of allocation)

- With small model, when **-QF** is specified
  char, int, short, enum type: in the order of RP3, VP, UP
  Pointer type: In the order of VP, UP , RP3
  long, float, double type: RP3 (lower), VP (higher)
- With small model, when **-QF** is not specified
  char, int, short, enum type: In the order of RP3, VP
  Pointer type: In the order of VP, RP3
  long, float, double type: RP3 (lower), VP (higher)
- With large model
  char, int, short, enum type: In the order of RP3, VP
  Pointer type: In the order of VVP
  long, float, double type: RP3 (lower), VP (higher)

(3) Location and order of storing automatic variables

- There are two types of automatic variables: automatic variables to be allocated to registers and ordinary automatic variables. The automatic variables to be allocated to registers are the ones that are declared with registers and the automatic variables when **-QV** is specified. They are allocated to register **_@KREGXX** as long as there are allocable registers and **_@KREGXX**. However, the automatic variables are allocated to **_@KREGXX** only when **-QR** is specified.
  The automatic variables allocated to registers and **_@KREGXX** are called register variables hereafter.
- For **_@KREGXX**, refer to **APPENDIX A LIST OF LABELS FOR saddr AREA**.
- The register variables are allocated after register arguments are allocated. Therefore, the register variables are allocated to registers when there are excess registers after the allocation of register arguments.
- The automatic variables not allocated to registers are allocated to stacks.
- The saving and restoring of registers and **_@KREGXX** to allocate automatic variables is performed on the function definition side.

(Order of allocating automatic variables)

- The order of allocating automatic variables to registers are the same as the order of allocating arguments. For the details, refer to the order of allocating arguments.
- The automatic variables allocated to **_@KREGXX** are allocated in the order of declaration.
- The automatic variables allocated to stacks are placed on the stack in the order of declaration.

The following shows an example of the interface above.

EXAMPLE 1

(C source)

```
void func0 (register int, int);
void main () {
        func0 (0x1234, 0x5678);
}
void func0 (register int p1, int p2) {
        register int r;
        int a;
        r = p2;
        a = p1;
}
```

(Output code) With large model, when **-QF** is specified and **-ZO** is not specified

```
@@CODE      CSEG
_main:
      movw  ax,#05678H   ;22136
      push  ax                  ;Arguments passed via stack
      movw  ax,#01234H   ;4660  ;The first argument is passed via register
      call  $!_func0            ;Function call
      pop   ax                  ;Arguments passed via stack
      ret
_func0:
      push  uup                 ;Save registers for register variables/arguments
      push  rp3                 ;
      push  vvp                 ;
      movw  rp3,ax              ;Allocate register arguments to rp3
      movw  ax,[sp+11]   ;p2    ;Argument p2 to be passed via a stack
      movw  up,ax               ;Register variable r (up)
      movw  vp,rp3              ;Register argument p1 (rp3) variable a (vp)
      pop   vvp                 ;Restores register for register variables/arguments
      pop   rp3
      pop   uup
      ret
```

11.7.3 noauto function call interface

(1) Passing arguments

(a) When the -ZO option is not specified (default)

- On the function call side, the arguments declared with registers and the ordinary arguments are passed in the same manner. The second and subsequent arguments are passed via a stack. The first argument is passed via a register or a stack (in the same manner as ordinary functions).
- For the location where the first argument is passed, refer to **Table 11-28**.

(b) When the -ZO option is specified

- Arguments are passed via registers. For the registers to be used, refer to **Table 11-13**.

(2) Location and order of storing arguments

- On the function definition side, all the arguments are allocated to registers.
- If there is an argument that cannot be allocated to a register, an error occurs.

(a) When the -ZO option is not specified (default)

- On the function definition side, the arguments passed via registers or stacks are copied to registers. Even when the arguments are passed via registers, the processing to copy the register is output because the register on the function call side (passing side) and the function definition side (receiving side) are different. For the registers allocated on the function definition side, refer to **Table 11-14**.
- The saving and restoring of the register to which arguments are allocated is performed on the function definition side.

(Order of allocation)

- The order is the same as an ordinary function with **-QF** specified.

(b) When the -ZO option is specified

- The locations where arguments are passed on the function call side and the function definition side are the same as the locations where arguments are allocated.
- The saving and restoring of registers to which arguments are allocated is performed before and after the function call.

(Order of allocation)

- The order is the same as for ordinary functions.

(3) Location and order of storing automatic variables

(a) When the -ZO option is not specified (default)

Automatic variables are allocated to registers and **_@KREGXX**. However, the automatic variables are allocated to **_@KREGXX** only when **-QR** is specified. For **_@KREGXX**, refer to **APPENDIX A LIST OF LABELS FOR saddr AREA**.

Automatic variables are allocated to registers when there are excess registers after the allocation of arguments. When **-QR** is specified, automatic variables are allocated also to **_@KREGXX**.

If an automatic variable cannot be allocated to registers and **_@KREGXX**, an error occurs.

The saving and restoring of the register and **_@KREGXX** to which automatic variables is allocated are performed in the function definition side.

(Order of allocation)

- The order of allocating automatic variables to registers are the same as the order of allocating arguments.
- The automatic variables allocated to **_@KREGXX** are allocated in the order of declaration.

(b) When the -ZO option is specified.

- Allocation cannot be performed because the automatic variables cannot be described.

The following shows an example of the interface above.

EXAMPLE

(C source)

```
noauto void func2 (int, int);
void main ( ) {
        func2 (0x1234, 0x5678);
}
noauto void func2 (int p1, int p2) {
        /* function body */
}
```

(Output code) With small model, when **-ZO** is specified

```
@@CODES        CSEG     BASE
_main:
       push    rp3,vp                    ;Save registers for arguments
       movw    rp3,#01234H     ;4660     ;Allocate arguments to rp3
       movw    vp,#05678H      ;22136    ;Allocate arguments to vp
       call      !_func2                 ;Function call
       pop     rp3,vp                    ;Restore registers for arguments
       ret
_func2:
       ret
```

(Output code) With small model, when **-ZO** is not specified

```
@@CODES        CSEG        BASE
_main:
       movw    ax,#05678H  ; 22136
       push    ax                    ;Arguments passed via stack
       movw    ax,#01234H  ; 4660    ;The first argument is passed via register
       call    !_func2               ;Function call
       pop     ax                    ;Arguments passed via stack
       ret
_func2:
       push    rp3,up                ;Save registers for arguments
       movw    rp3,ax                ;Allocate arguments to rp3
       movw    ax,[sp+7]             ;Argument passed via stack received by register
       movw    up,ax                 ;Allocate arguments to up
       pop     rp3, up               ;Restore registers for arguments
       ret
```

11.7.4 norec function call interface

(1) Passing arguments

(a) When the -ZO option is not specified (default)

On the function call side, arguments are passed via registers and **_@NRARGX**. For the registers, refer to **Table 11-17 Registers Used for norec Function Arguments: Passing Side** (**Without -ZO**).

(b) When the -ZO option is specified

On the function call side, arguments are passed via a register and **_@NRARGX**. If the arguments cannot be passed via registers any more, they are passed only via **_@NRARGX** instead of via registers. Arguments are never passed via registers and **_@NRARGX** together.

(2) Location and order of storing arguments

- On the function definition side, all the arguments are allocated to registers and **_@NRARGX**. However, arguments are allocated to **_@NRARGX** only when **-QR** is specified. For **_@NRARGX**, refer to **APPENDIX A LIST OF LABELS FOR saddr AREA**.
- If there is an argument that cannot be allocated to registers and **_@NRARGX**, an error occurs.

(a) When the -ZO option is not specified (default)

- On the function definition side, the arguments passed via registers are copied to registers. Even when the arguments are passed via registers, copying the register is necessary because the register on the function call side (passing side) and the function definition side (receiving side) are different.

  When the arguments are passed via **_@NRARGX**, the locations where arguments are passed are the same as the locations where arguments are allocated.

  If the arguments cannot be passed via registers any more, they are passed also via **_@NRARGX**. Arguments are passed via registers and **_@NRARGX** together.

  The saving and restoring of the register to which arguments are allocated is performed in the function definition side. For the location of storing arguments, refer to **Table 11-18 Registers Used for norec Function Arguments: Receiving Side** (**Without -ZO**).

Table 11-31. List of Registers Passing/Receiving norec Arguments (When -ZO Is Not Specified)

Type \ Model	Small Model, Medium Model[Note 1]	Large Model[Note 2]
The first argument is **char** type	Passed via C, DE, RP2 Received via R6, R7, VP, UP	Passed via C, TDE, RP2 Received via R6, R7, VVP, UP
The first arguments is not **char** type	Passed via AX, DE, RP2 Received via RP3, VP, UP	Passed via AX, TDE, RP2 Received via RP3, VVP, UP

Notes 1. With the medium model, the function pointer (3 bytes) cannot be used via a register. When **-QR** is specified, however, it can be passed via _**@NRARGX**.

2. With the large model, only one pointer (3 bytes) can be passed/received via a register. When **-QR** is specified, however, it can be passed/received also via _**@NRARGX**.

(Order of allocation)

- With small model, medium model
  - **char**, **int**, **short**, **enum** type: If there is **long**, **float**, **double** type argument, in the order of UP, RP3, VP
    If there is no **long**, **float**, **double** type argument, in the order of RP3, UP, VP
  - Pointer type: In the order of UP, VP, RP3
  - **long**, **float**, **double** type: RP3 (lower), VP (higher)
- With large mode
  - **char**, **int**, **short**, **enum** type: If there is **long**, **float**, **double** type argument, in the order of UP, RP3, VP
    If there is no **long**, **float**, **double** type argument, in the order of RP3, UP, VP
  - Pointer type: VVP
  - **long**, **float**, **double** type: RP3 (lower), VP (higher)

(b) When the -ZO option is specified

- The same as the **noauto** function call interface

(3) Location and order of storing automatic variables

(a) When the -ZO option is not specified

The automatic variables are allocated to registers and **_@NRARGX** as long as there are allocable registers and **_@NRARGX**. If there is no allocable register any more, they are allocated to **_@NRATXX**.

However, automatic variables are allocated to **_@NRARGX** and **_@NRATXX** only when **-QR** is specified.

For **_@NRATXX**, refer to **APPENDIX A LIST OF LABELS FOR saddr AREA**

If there is an automatic variable that cannot be allocated to registers, **_@NRARGX** and **_@NRATXX**, an error occurs.

The saving and restoring of registers to which automatic variables are allocated is performed on the function definition side.

(Order of allocating automatic variables)

- The order of allocating automatic variables to registers is the same as the order of allocating **noauto** function arguments. For details, refer to **11.7.3 noauto function call interface**.
- The automatic variables allocated to **_@NRATXX** are allocated in the order of declaration.

(b) When the -ZO option is specified

- The automatic variables are allocated to registers as long as there are allocable registers. If there are no more allocable registers, they are allocated to **_@NRATXX**.
- Automatic variables are allocated to **_@NRATXX** only when **-QR** is specified. For **_@NRATXX**, refer to **APPENDIX A LIST OF LABELS FOR saddr AREA**.
- The automatic variables are allocated after arguments are allocated. Therefore, the automatic variables are allocated to registers when there are excess registers after the allocation of arguments.
- If there is an automatic variable that cannot be allocated to a register and **_@NRATXX**, an error occurs.
- The saving and restoring of registers to allocate automatic variables is performed on the function definition side.

(Order of allocating automatic variables)

- The order of allocating registers to automatic variables is the same as the order of allocating **noauto** function arguments. For details, refer to **11.7.3 noauto function call interface**.
- The automatic variables allocated to **_@NRARGX** and **_@NRATXX** are allocated in the order of declaration.

EXAMPLE

(C source)

```
    norec  void func (int);
    void main (void) {
            func (0x34);
    }
    norec  void func (int p1) {
            int a;
            a = p1;
    }
```

(Output code) With small model, when **-QX2** and **-ZO** are specified

```
          @@CODES CSEG
_main:
          push    rp3             ;Save registers for arguments
          movw    rp3,#034H; 52   ;Allocate arguments to RP3
          call    $!_func3        ;Function call
          pop     rp3             ;Restore registers for arguments
          ret
_func:
          push    vvp             ;Save the automatic variable register
          movw    vp,rp3          ;a = p1
          pop     vvp             ;Restore the automatic variable register
          ret
```

(Output code) With small model, when **-QX2** and **-ZO** is not specified

```
@@CODE            CSEG
_main:
          movw    ax,#034H ;52          ;Transfers the argument at AX
          call    $!_func               ;Function call
          ret
_func:
          push    uup                   ;Save the automatic variable register
          push    rp3                   ;Save registers for arguments
          movw    rp3,ax                ;Store argument in RP3
          movw    up,rp3                ;a = p1
          pop     rp3                   ;Restore registers for arguments
          pop     uup                   ;Restore the automatic variable register
          ret
```

11.7.5 Pascal function call interface

The difference between this function interface and other function interfaces is that the correction of stacks used for loading of arguments when a function is called is done by the function side that was called, rather than the function caller. All other points are the same as the function attributes specified at the same time.

[Area to which arguments are allocated]
[Sequence in which arguments are allocated]
[Area to which automatic variables are allocated]
[Sequence in which automatic variables are allocated]

- If the **noauto** attribute is specified at the same time, the features are the same as when a **noauto** function is called (Refer to **11.7.3 noauto function call interface**).
- If the **noauto** attribute is not specified at the same time, the features are the same when an ordinary function is called (Refer to **11.7.2 Ordinary function call interface**).

(C source)

```
_ _pascal void func0 (register int, int);
void main ()
{
        func0 (0x1234, 0x5678);
}
_ _pascal void func0 (register int p1, int p2)
{
        register int r;
        int a;
        r = p2;
        a = p1;
}
```

(Output code)

With small model (when **-QF** option is specified)

```
_main:
; line 4 : func0(0x1234, 0x5678);
        movw ax,#05678H ; 22136
        push ax                         ; Argument is passed via a stack
        movw ax,#01234H ; 4660          ; The first argument is passed via a register
        call !_func0                    ; Function call
                                        ; Stack is not corrected here
        ret
; line 6 : _ _pascal void func0(register int p1, int p2)
; line 7 : {
_func0:
        push rp3,up                     ; Saves the register for register variables
                                        ; or register arguments
        movw rp3,ax                     ; Allocates a register argument to rp3
        push ax                         ; Reserves the area for automatic variable a
; line 8 : register int r;
; line 9 : int a;
; line 10 : r = p2;
        movw ax,[sp+9]; p2              ; Argument p2 is passed via stack
        movw up,ax                      ; Register variable up
; line 11 : a = p1;
        movw ax,rp3                     ; Register argument rp3
        movw [sp+0],ax ; a              ; Automatic variable a
        pop ax                          ; Releases the area for automatic variable a
        pop rp3,up                      ; Restores the register for register variables
                                        ; or register arguments
        pop hl                          ; Obtains the return address
        incg sp                         ;
       pop ax                           ; The stack consumed by arguments passed via a
                                        ; stack is corrected
        br hl                           ; Branch to the return address
```

(C source)
With large model

```
_ _pascal noauto void func2(int, int);
void main ()
{
        func2(0x1234, 0x5678);
}
_ _pascal noauto void func2(int p1, int p2)
{
      ...
}
```

(Output code)
With large model

```
  _main:
  ; line 4 : func2(0x1234, 0x5678);
          movw ax,#05678H ; 22136
          push ax                          ; Argument is passed via a stack
          movw ax,#01234H ; 4660           ; The first argument is passed via a register
          call $!_func2                    ; Function call
                                           ; Stack is not corrected here

ret
; line 6 : _ _pascal noauto void func2(int p1, int p2)
; line 7 : {
_func2:
        push uup                 ; Saves the register for arguments
        push rp3                 ; Saves the register for arguments
        movw rp3,ax              ; Allocates a register argument to rp3
        movw ax,[sp+8]           ; Argument passed via a stack and received by a register
        movw up,ax               ; Allocates an argument to up
        ...
        pop rp3                  ; Restores the register for arguments
        pop uup                  ; Restores the register for arguments
        pop whl                  ; Obtains the return address
        pop ax                   ; The stack consumed by arguments passed via a stack is corrected
     br whl                      ; Branch to the return address
```

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 RA78K4 assembler package and this C compiler is described in the following order.

(1) Calling assembly language routines from C language
(2) Calling C language functions from assembly language
(3) Referencing variables defined in C language
(4) Referencing variables defined in assembly language on the C language side
(5) Cautions

12.1 Accessing Arguments/Automatic Variables

The procedure for accessing arguments and automatic variables of this C compiler is described below.

- On the function call side, register arguments are passed in the same way as ordinary arguments.
  The first argument uses the following registers and stacks, and subsequent arguments are passed via stacks.

Table 12-1. Passing Arguments (Function Call Side)

Type	Passing Location (First Argument)	Passing Location (Second and Later Arguments)
1-byte, 2-byte integer	AX	Stack passing
3-byte integer	WHL (Stack passing in case of small model)	Stack passing
4-byte integer	AX, RP2	Stack passing
Floating-point number	AX, RP2	Stack passing
Others	Stack passing	Stack passing

Remark 1- to 4-byte data includes structures and unions.

- On the function definition side, arguments passed via a register or stack are stored in the argument allocation location.
  Register arguments are copied to a register or **saddr** area (**_@KREGxx**).
  Even when passing is done via a register, the registers on the function call side (passing side) and the function definition side (receiving side) differ, and therefore register copying is performed.
  Ordinary arguments passed via a register are placed on a stack on the function definition side.
  If passing is done via a stack, the passing location simply becomes the argument allocation location.
  Saving and restoring of registers that allocate arguments is performed on the function definition side.
- The arguments of functions and the values of automatic variables declared inside functions are stored in the following registers, **saddr** areas, or stack frames using an option.
  The base pointer used when storing in a stack frame uses the **UP** register.

Table 12-2. List of Storing Arguments/Automatic Variables (Inside Called Function)

Option	Argument/auto Variable	Storage Location	Priority Level
-QV (register allocation option)	Declared argument or automatic variable	• With small or medium model RP3, VP, UP (only when **-QF** is specified) • With large model RP3, VVP, UUP (only when **-QF** is specified)	Although the allocation order may vary depending on the number of references, the priority level is determined basically by the following rules. <1> With small or medium model • When **-QF** is specified **char**, **int**, **short**, **enum** type: In the order of UP, RP3, VP (if a **long**, **float**, or **double** type argument exists) In the order of RP3, UP, VP (if a **long**, **float**, or **double** type argument does not exist) Pointer type: In the order of UP, VP, RP3 **long**, **float**, or **double** type: RP3 (lower), VP (higher) • When **-QF** is not specified **char**, **int**, **short**, **enum** type: In the order of RP3, VP Pointer type: In the order of VP, RP3 **long**, **float**, or **double** type: RP3 (lower), VP (higher) <2> With large model • When **-QF** is specified **char**, **int**, **short**, **enum** type: In the order of UP, RP3, VP (if the **long**, **float**, or **double** type argument exists) In the order of RP3, UP, VP (if the **long**, **float**, or **double** type argument does not exist) Pointer type: In the order of UUP, VVP, **long**, **float**, or **double** type: RP3 (lower), VP (higher) • When **-QF** is not specified **char**, **int**, **short**, **enum** type: In the order of RP3, VP Pointer type: In the order of VVP **long**, **float**, or **double** type: RP3 (lower), VP (higher)
-QR	register declared automatic variable	• With small or medium model RP3, VP, UP (only when **-QF** is specified) • With large model RP3, VVP, UUP (only when **-QF** is specified) • Automatic variable _@KREGxx	
-QRV	Declared argument or automatic variable	• With small or medium model RP3, VP, UP (only when **-QF** is specified) • With large model RP3, VVP, UUP (only when **-QF** is specified) • Automatic variable _@KREGxx	
Default	Declared argument, automatic variable	Stack frame	Order of appearance

The following example shows the function call.

(C source: Large model with **-QRF**)

```
void func0(register int, int);
void main()
{
        func0(0x1234, 0x5678);
}
void func0(register int p1, int p2)
{
        register int r;
        int a;
        r = p2;
        a = p1;
}
```

(Output assembler source)

```
          PUBLIC _func0
          PUBLIC _main

@@CODE  CSEG
_main:
          movw   ax,#05678H    ; 22136
          push   ax                        ; Argument is passed via a stack
          movw   ax,#01234H    ; 4660      ; The first argument is passed via a register
          call   $!_func0                  ; Function call
          pop    ax                        ; Argument is passed via a stack
          ret
_func0:
          push   uup                       ; Saves the register for arguments
          push   rp3
          movw   rp3,ax                    ; Allocates register arguments p1 to rp3.
          push   ax
          movw   ax,[sp+10]    ; p2        ; Argument p2 passed via a stack is allocated to up
          movw   up,ax
          movw   ax,rp3                    ; Register argument p1 is assigned
          movw   [sp+0],ax     ; a         ; to automatic variable a
          pop    ax
          pop    rp3                       ; Restores the register for arguments
          pop    uup
          ret
        END
```

12.2 Storing Return Values

Return values during function calls are stored in registers and carry flags.
The storage locations of return values are shown in the table below.

Table 12-3. Storage Location of Return Values

Type	Small Model	Medium Model	Large Model
1-byte integer 2-byte integer	BC	BC	BC
4-byte integer	BC (lower), RP2 (higher)	BC (lower), RP2 (higher)	BC (lower), RP2 (higher)
Pointer	BC	BC (data pointer) WHL (function pointer)	TDE
Structure, union	BC (start address of structure or union copied to function-specific area)	BC (start address of structure or union copied to function-specific area)	TDE (start address of structure or union copied to function-specific area)
1 bit	CY	CY	Y
Floating-point number	BC (lower), RP2 (higher)	BC (lower), RP2 (higher)	C (lower), RP2 (higher)

12.3 Calling an Assembly Language Routine from C

By default (when **-ZO** is not specified), the first argument is passed via a register (refer to **Table 11-28 Location Where First Argument Is Passed (On Function Call Side)**). When **-ZO** is specified, all the arguments are passed via stacks. This example shows a case in which **-ZO** is not specified and the default optimization option (**-QCFHJLVW**) is specified.

How to call an assembly language routine from C is explained in the following order.

- Calling an assembly language routine function (C source)
- Saving and restoring the information of an assembly language routine (Assembler source)

(1) Calling an assembly language routine function (C source)

An example of a C language program to call an assembly language routine is shown below.

```
extern int FUNC (int, long);            /* function prototype */

void main ( ) {
        int    i, j;
        long   l;

        i = 1;
        l = 0x54321;
        j = FUNC (i, l);                /* function call */
}
```

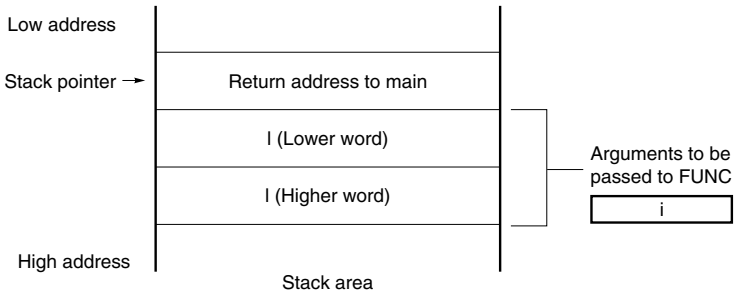
In this program example, how the two programs interface with each other at execution time and the flow of control between the two are explained below.

<1> The arguments are passed from the **main** function to the **FUNC** function.
The compiler assigns an argument to the register or outputs the code to be placed on the stack.

<2> Control is transferred to the **FUNC** function by the **CALL** instruction.
The compiler outputs the **CALL** instruction.

The stack area immediately after the transfer of control to the **FUNC** function in the above example looks like this.

Figure 12-1. Stack Area After Call



(2) Saving and restoring the information of assembly language routine (assembler source)

The **FUNC** function called from **main** executes the following processes.

<1> (Saving the base pointer)[Note]
<2> Saving the work registers
<3> (Copying the contents of the stack pointer (SP) to the base pointer (UUP/UP))[Note]
(With large model copying to UUP, with small/medium model copying to UP)
<4> Processing the body of **FUNC**
<5> Setting the return value
<6> Restoring the saved registers
<7> Returning control to **main**

Note Since this example shows a case in which the default optimization option is used, SP is used for the stack manipulation. Therefore, the processing in <1> and <3> is not necessary. When the **-QF** option is not specified, however, the processing in <1> and <3> is necessary.

An example of an assembly language program is shown below.

```
@@DATA DSEG
_DT1:  DS      (2)
_DT2:  DS      (4)

@@CODE CSEG
_FUNC:
        push   uup            ;save work registers ····· <2>
        push   rp3
        push   vvp
        movw   up,ax          ;arg1
        movw   ax,[sp+11]     ;arg2
        movw   rp3,ax
        movw   ax,[sp+13]     ;arg2
        movw   vp,ax
        movw   !!_DT1,up      ;move 1st argument(i)
        movw   !!_DT2,rp3     ;move 2nd argument(l)
        movw   !!_DT2+2,vp
        movw   bc,#0AH ; 10   ;set return value ····· <5>
        pop    vvp            ;restore work registers
        pop    rp3
        pop    uup
        ret
```

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.

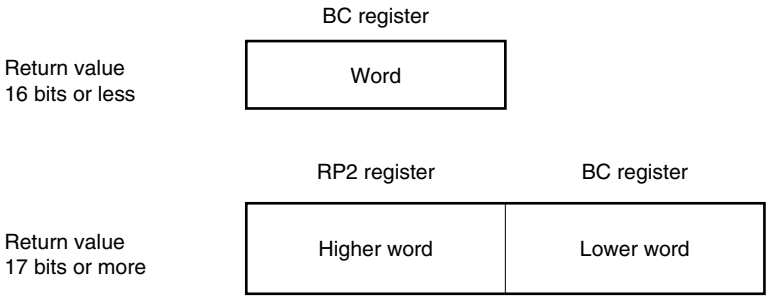
<1> Saving the base pointer
In this example, SP is used because a case in which the **-QF** option is specified is shown.
Therefore, the saving of the base pointer is not performed.

<2> Saving the work registers
In a program created by the C compiler, other functions are called without saving the registers for storing variables. For this reason, if the values of these variables are to be changed by the function to be called, the register values must be saved beforehand.
If no register variable is used by the caller, the contents of the work registers need not be saved.

<3> Copying the contents of the stack pointer (SP) to the base pointer (UUP/UP)
The value of the stack pointer (SP) will be changed by a PUSH or POP instruction inside the function. For this reason, the stack pointer must be saved to the register 'UUP (large model)' or 'UP (small/medium model)' to use it as the base pointer of the arguments.
SP is used in this example. Therefore, copying to the base pointer is not necessary.

<4> Processing the body of **FUNC**
On completion of steps <1> through <3> above, the body (declarations and statements) of the **FUNC** function is processed.

<5> Setting the return value
If **FUNC** has any value to return, the return value is set in the BC register or RP2 and BC registers; otherwise, nothing is set in these registers. For register to store the return value, refer to **Table 11-27 Location of Storing Return Value**.

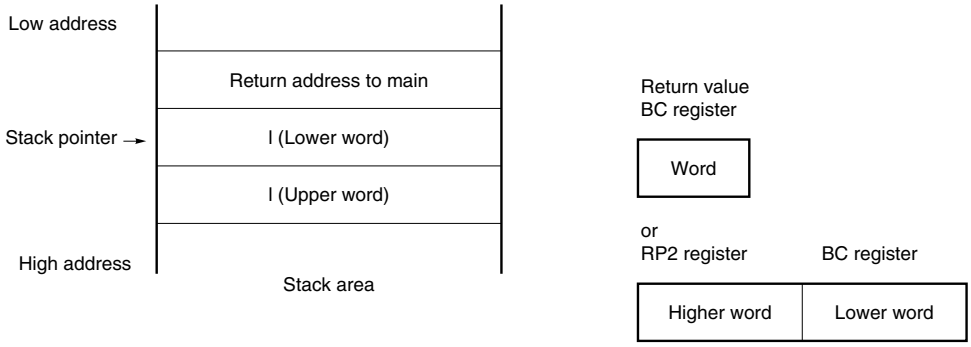


<6> Restoring the saved registers

The saved contents of the base pointer and work registers are restored.

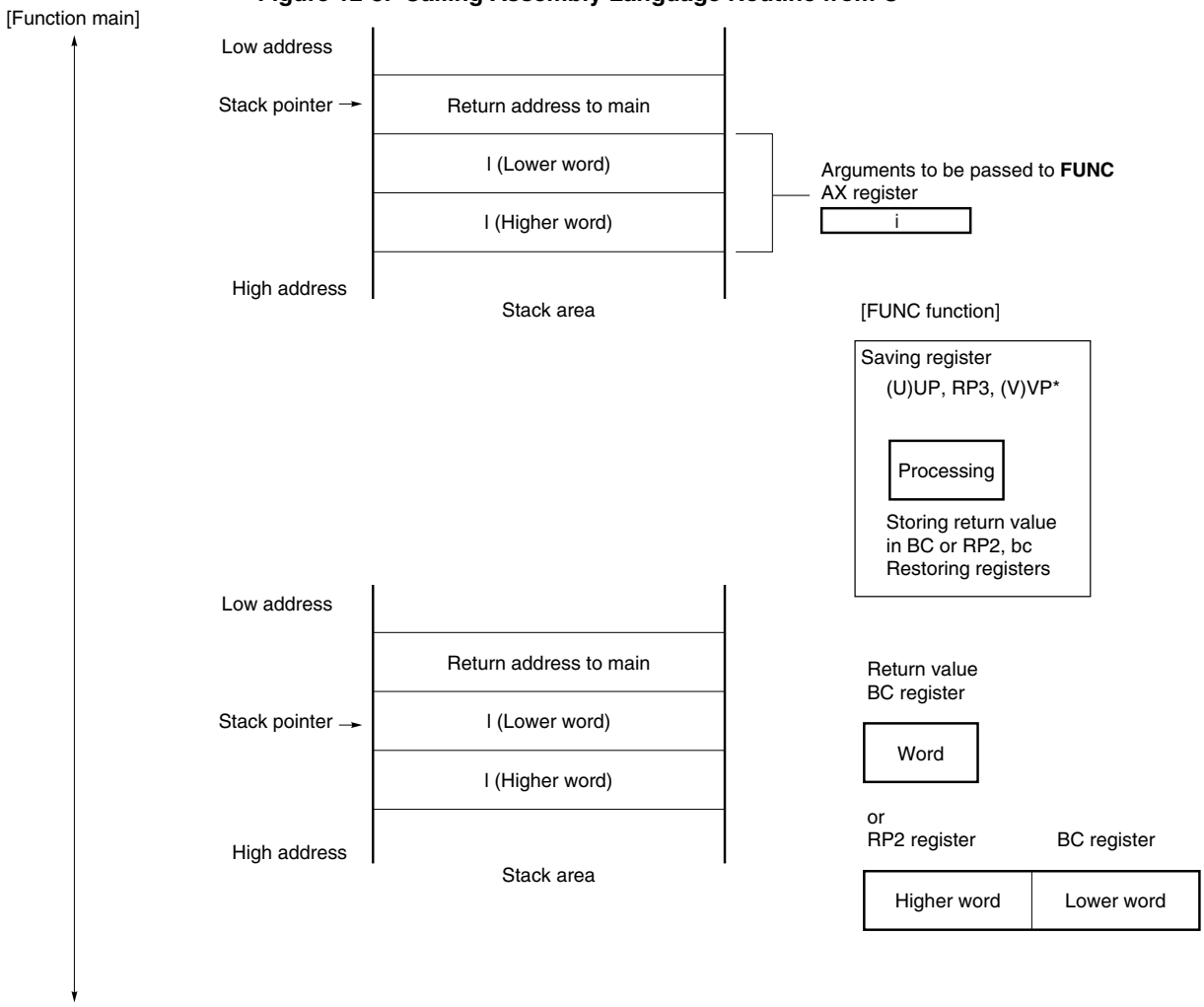
<7> Returning control to **main**

Figure 12-2. Stack Area After Return



The procedure for calling an assembly language from C and the processing of the assembly language routine are illustrated in **Figure 12-3**.

Figure 12-3. Calling Assembly Language Routine from C



12.4 Calling C Language Routine from Assembly Language Routine

(1) Calling a C language function from assembly language (assembler source)

A function written in C language can be called from an assembly language routine by the following procedure.

<1> Copy the first argument to a register and place the remaining arguments of the function on the stack. (Refer to **Table 11-28 Location Where First Argument Is Passed (On Function Call Side)**).
<2> Call the C language function.
<3> Change the value of the stack pointer (SP) for the number of bytes of the arguments (except the number of bytes of the first argument).
<4> Reference the return value of the C language function (stored in the BC or RP2 and BC registers).

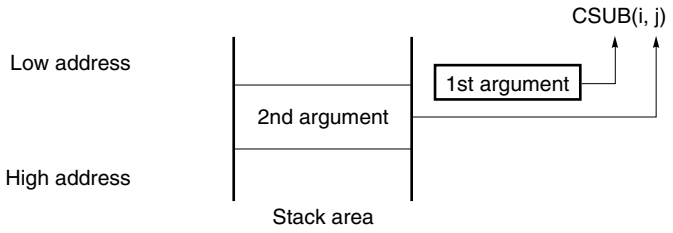
An example of an assembly language program is shown below.

```
            NAME      FUNC2
            EXTRN     _CSUB
            PUBLIC    _FUNC2

CODE2       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> Placing the arguments on the stack

If there are two or more arguments, the second and subsequent arguments are placed on the stack. The arguments are passed as shown in **Table 12-4**. When the **-ZO** option is specified on the C source side, however, all the arguments are placed on the stack.

Figure 12-4. Placing Arguments of Stack



<2> Calling the C language function

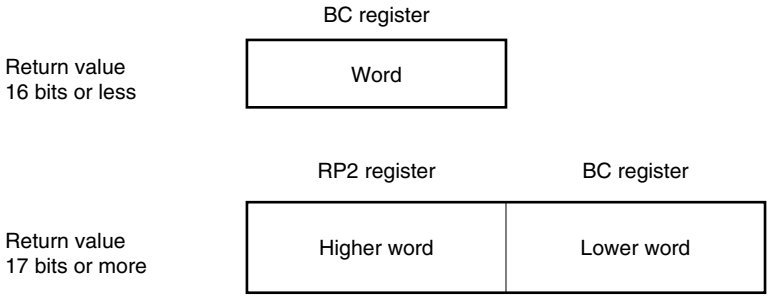
The **CALL** instruction must be used to call a C language function.

<3> Changing the value of the stack pointer (SP)

The value of the stack pointer (SP) must be changed for the number of bytes of the arguments placed on the stack. In this example, because arguments of 2 bytes are to be passed, 2 is added to the value of the stack pointer. (POPped in the example)

<4> Referring to the return value (BC or RP2 and BC)

The return value (in the BC register or RP2 and BC register) from C language function is stored as follows.



12.5 Referencing Variables Defined by Other Languages

(1) How to refer to C-defined variables

To refer to external variables that have been defined in a C language program in an assembly language routine, the variables must be declared as **extern** (external) in the C language program.

[Example of program] With large model

(C source)

```
extern void subf () ;

char   c = 0 ;
int    i = 0 ;
void main () {
       subf () ;
}
```

In the RA78K4 assembler, the C-defined variables must be described as follows.

(Assembler source)

```
        PUBLIC      _subf
        EXTRN       _c
        EXTRN       _I

@@CODE CSEG
_subf:
        MOV         A, #04H
        MOV         !!_c, A
        MOVW        AX, #07H
        MOVW        !!_i, AX
        RET
        END
```

(2) How to refer to assembler-defined variables from C

To refer to variables that have been defined in assembly language from C, the variables must be described in the C language program as follows.

[Example of C language program] With large model

(C source)

```
    extern char c ;
    extern int    i ;

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

In the RA78K4 assembler, the assembler-defined variables must be described as follows.

```
        NAME     ASMSUB

        PUBLIC   _c
        PUBLIC   _i

ABC     CSEG
_c:     DB 0
_I:     DW 0

        END
```

12.6 Other Important Hints

(1) "_" (underscore)

With this C compiler, "_" (underscore: ASCII code "5FH") is prefixed to each external variable or reference name. In the following C program example, "j = FUNC(i, l);" is interpreted as a reference to the external name "_FUNC".

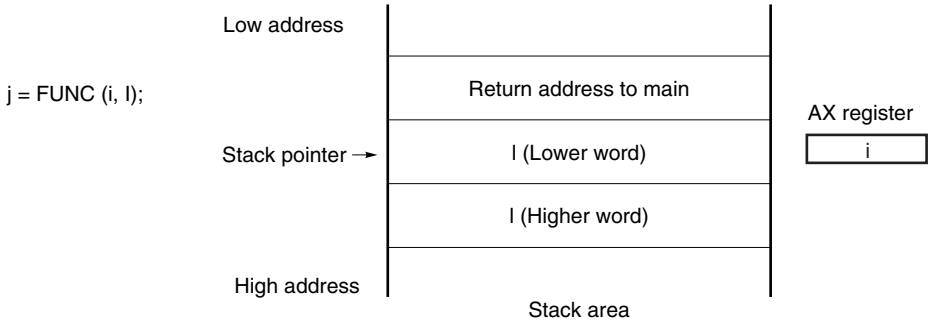
```
extern int FUNC(int, long);        / * function prototype */

void main ( ) {
       int    i, j;
       long   l;

       i = 1;
       l = 0x54321;
       j = FUNC (i, l);            / * function call */
}
```

In the RA78K4, the routine name must be described as "_FUNC".

(2) Placement of arguments on the stack

Arguments are placed on the stack in sequence from the last to the first argument in the direction from the higher to the lower address. When **-ZO** is not specified on the C source side, the first argument is passed via a register.

Figure 12-5. Placement of Arguments on Stack

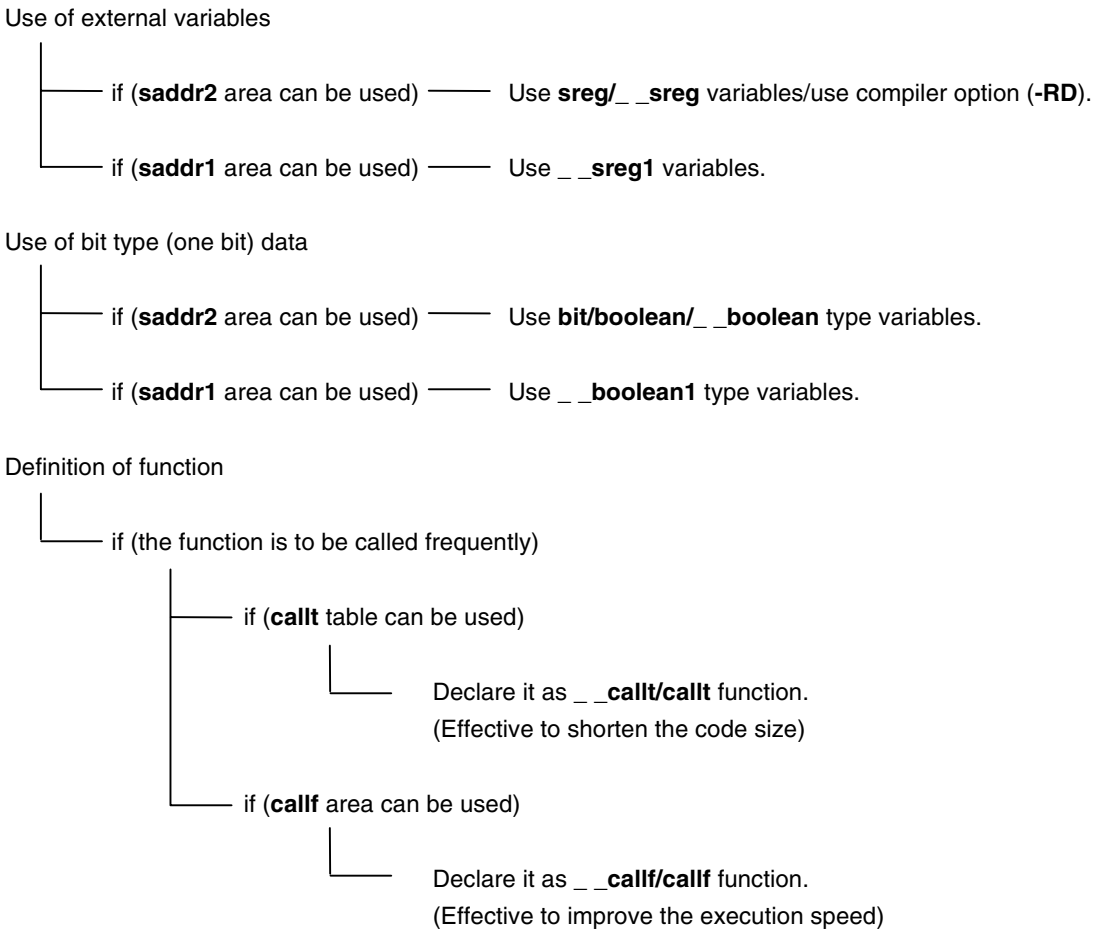


CHAPTER 13 EFFECTIVE UTILIZATION OF COMPILER

This chapter introduces how to effectively use this C compiler.

13.1 Efficient Coding

When developing 78K/IV Series microcontroller-applied products, efficient object generation may be realized with this C compiler by utilizing the **saddr1/2** area, **callt** table, or **callf** area of the device.



(1) Using external variables

When defining an external variable, specify the external variable to be defined as a **sreg/_ _sreg** variable if the **saddr2** 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.)

When **saddr1** area as well as **saddr2** area can be used, the similar effect can be achieved by specifying the external variable to be defined as _ _**sreg1** variable.

Definition of **sreg/_ _sreg** variable:	`extern sreg` int variable-name `;` `extern_ _sreg int` variable-name `;`

Remark Refer to **11.5 (3) How to use the saddr area**.

(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 a **bit/boolean/_ _boolean** type variable. Because **saddr** area is used as well as the **sreg** variable, the codes can be shortened and the execution speed can be improved.

When **saddr1** area as well as **saddr2** area can be used, the similar effect can be achieved by specifying the external variable to be defined as a _ _**boolean1** type variable.

Declaration of **bit/boolean** type variable:	`bit` variable-name `;`
	`boolean` variable-name `;`
	`_ _boolean` variable-name `;`

Remark Refer to **11.5 (7) bit type variables**.

(3) Function definitions

For a function to be called over and over again, object code should be shortened or a structure which allows calling at high speeds should be provided. If the **callt** table can be used for functions to be called frequently, such functions should be defined as **callt** functions. Likewise, if the **callf** area can be used for functions to be called frequently, such functions should be defined as **callf** functions. The **callf** functions can be called faster than ordinary function calls with shorter codes because the **callf** functions are called using the **callf** area of the device. The **callt** functions are effective when codes needs to be shortened because the **callt** functions use the **callt** area of the device and are called with shorter code than **callf**.

Definition of **callt** function:
```
callt int tsub() {
        :
}
```
Definition of **callf** function:
```
callf int tsub()
        :
}
```

Remark Refer to **11.5 (1) callt function** and **11.5 (15) callf function**.

In addition to the use of the areas shown above, objects that do not need modification of the C source by compiling with the optimization option can be generated. For the effect of each **-Q** suboption, refer to the **CC78K4 C Compiler Operation User's Manual (U15557E)**.

(4) Optimization option

The optimization options that emphasize the object code size the most is as follows.

[Object code is emphasized the most]

```
-QX3
```

Further shortening of the code size and improvement of the execution speed is possible by adding _ _**sreg** or _ _**sreg1** to variables. However, this is restricted to the cases when **saddr2** area or **saddr1** area can be used. When the areas have no more space and cannot be used, a compilation error occurs.

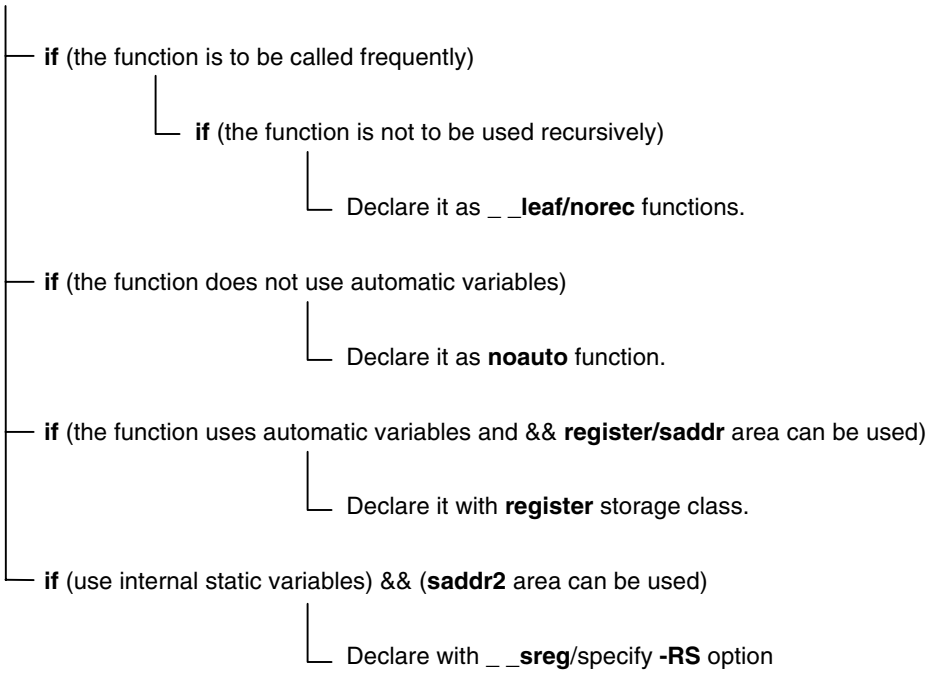
If execution speed is also highly emphasized, specify the **-QX2** default.

If the code size is smaller than **-QX3**, **-QX4** can be specified. However, there are restrictions during debugging.

In addition, the object efficiency can be improved by adding the extended functions supported by this compiler to the C source.

(5) Using extended functions

- Definition of function



- Functions not used recursively

  Of the functions to be called over and over again, the ones which are not used recursively should be defined as _ _**leaf/norec** functions. **norec** functions become functions that do not have preprocessing/ postprocessing (stack frame). Therefore, the object code can be shortened and the execution speed can be improved compared to the ordinary functions.

  Remark For the definition of the **norec** function (`norec int rout ()...`), refer to **11.5 (6) norec function** and **11.7.4 norec function call interface**.

- Functions that do not use automatic variables
  Functions that do not use automatic variables should be defined as **noauto** functions. These functions will not output code for stack frame generation and their arguments will be passed to registers as much as possible. These functions help shorten object code and improve program execution speed.

  Remark For the definition of the **noauto** function (`noauto int sub1 (int i)...`), refer to **11.5 (5) noauto functions** and **11.7.3 noauto function call interface**.

- Functions that use automatic variables
  If the **saddr2** area can be used for a function that uses 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 one using memory, and object code can be shortened as well.

  Remark For the definition of the **register** variable (`register int i; ...`), refer to **11.5 (2) Register variables**.

- Functions that use internal static variables
  If the **saddr2** 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.
  When **saddr1** area can be used as well as **saddr2** area, the same effect can be achieved by declaring the function with _ _**sreg1**.

  Remark Refer to **11.5 (3) How to use saddr area**.

In addition, the code efficiency and the execution speed can be improved by the following methods.

- Use of SFR name (or SFR bit name).

```
#pragma sfr
```

- Use of _ _**sreg/**_ _**sreg1** declaration for bit fields that 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

With the CC78K4, addresses in the **saddr2** area are referenced by the following label names. For this reason, the same names as these label names cannot be used in the C source program or assembler source program.

For the areas of Section A.1 to A.3, any consecutive 32-byte area of **saddr2** area (F) FD20H to (F) FDFFH is used. The allocation addresses are determined at linking.

Remark (F)FDXXH indicates the address where _**@NRARG0** is allocated, and F is added to the higher 4 bits at the location 1024K (0FH: Compiler option **-CS15**).

A.1 Arguments of norec Functions

Label Name	Address
_@NRARG0	(F)FDXXH
_@NRARG1	_@NRARG0 + 1H
_@NRARG2	_@NRARG0 + 2H
_@NRARG3	_@NRARG0 + 3H
_@NRARG4	_@NRARG0 + 4H
_@NRARG5	_@NRARG0 + 5H
_@NRARG6	_@NRARG0 + 6H
_@NRARG7	_@NRARG0 + 7H

A.2 Automatic variables of norec Functions

Label Name	Address
_@NRAT00	_@NRARG0 + 8H
_@NRAT01	_@NRARG0 + 9H
_@NRAT02	_@NRARG0 + AH
_@NRAT03	_@NRARG0 + BH
_@NRAT04	_@NRARG0 + CH
_@NRAT05	_@NRARG0 + DH
_@NRAT06	_@NRARG0 + EH
_@NRAT07	_@NRARG0 + FH

A.3 Register Variables

Label Name	Address
_@KREG00	_@NRARG0 + 10H
_@KREG01	_@NRARG0 + 11H
_@KREG02	_@NRARG0 + 12H
_@KREG03	_@NRARG0 + 13H
_@KREG04	_@NRARG0 + 14H
_@KREG05	_@NRARG0 + 15H
_@KREG06	_@NRARG0 + 16H
_@KREG07	_@NRARG0 + 17H
_@KREG08	_@NRARG0 + 18H
_@KREG09	_@NRARG0 + 19H
_@KREG10	_@NRARG0 + 1AH
_@KREG11	_@NRARG0 + 1BH
_@KREG12	_@NRARG0 + 1CH
_@KREG13	_@NRARG0 + 1DH
_@KREG14	_@NRARG0 + 1EH
_@KREG15	_@NRARG0 + 1FH

APPENDIX B LIST OF SEGMENT NAMES

This chapter explains all the segments that the compiler outputs and their locations.
(1) to (3) shows the options and re-allocation attributes used in the table.

(1) Option

–MS:	Small model
–MM:	Medium model
–ML:	Large model
–**CS0**:	Location 00H
–CS15:	Location 0FH

(2) Relocation attribute of CSEG

CALLT0:	Allocates the specified segment in the address 40H to 7FH with the start address of a multiple of 2.
BASE:	Allocates the specified segment in the address 80H to 0FCFFH.
AT absolute expression:	Allocates the specified segment in an absolute address (within 0H to 0FCFFH, 10000H to 0FFFFFH)[Note].
FIXED:	Allocates the start address of the specified segment in the address 800H to 0FFFH.
FIXEDA:	Allocates the start address of the specified segment in the address 800H to 0FFFH and the end within 0FCFFH.
PAGE:	Allocates the specified segment in the address xxx00H to xxxFFH (within 0FFFFFH).
PAGE64K:	Allocates the specified segment not to extend over the 64 KB boundary (within 0H to 0FCFFH, 10000H to 0FFFFFH)[Note].
UNIT/without specification:	Allocates the specified segment to a given location (within 80H to 0FCFFH, 10000H to 0FFFFFH)[Note].
UNITP:	Allocates the specified segment to a given location with the start address in an even address (80H to 0FCFFH, 10000H to 0FFFFFH)[Note].

Note The range can be changed by specifying the **–CS** option.

(3) Re-allocation attributes of DSEG

SADDR:	Allocates the specified segment to **saddr1** area (saddr1 area: 0FE00H to 0FEFFH)[Note]
SADDR2:	Allocates the specified segment to **saddr2** area (saddr2 area: 0FD20H to 0FDFFH)[Note]
SADDRP:	Allocates the specified segment starting from an even address in **saddr1** area.
SADDRP2:	Allocates the specified segment starting from an even address in **saddr2** area.
SADDRA:	Allocates the specified segment to a given area in **saddr** area (**saddr** area: **saddr1** area/**saddr2** area).
AT absolute expression:	Allocates the specified segment to an absolute address.
UNIT/without specification:	Allocates the specified segment to a given location (within the memory area name “RAM”)[Note].
UNITP:	Allocates the specified segment to a given location starting from an even address (within the memory area name ‘RAM’)[Note].
PAGE:	Allocates the specified segment to a given location between XXXX00H to XXXXFFH (within 0FFFFFH)[Note].
PAGE64K:	Allocates the specified segment not to extend over the 64 KB boundary (within 0H to 0FCFFH, 10000H to FFFFFH)[Note].

Note The range can be changed by specifying the **–CS** option (the address may differ depending on the target device. For details, refer to the user's manual of the target device used).

B.1 List of Segment Names

B.1.1 Program area and data area

(1) With small model (when –MS is specified)

Section Name	Segment Type	Relocation Attribute	Description
@@BASE	CSEG	BASE	Segment for **callt** function and **interrupt** function
@@VECTnn	CSEG	AT nnH	Segment for interrupt vector table
@@CODES	CSEG	BASE	Segment for ordinary function codes
@@CNSTS	CSEG	BASE	Segment for **const** variables
@@CALFS	CSEG	FIXEDA	Segment for **callf** function
@@CALT	CSEG	CALLT0	Segment for table for **callt** function
@@RSINIT	CSEG	BASE	Segment for initialization data (with initial value)
@@RSINIS	CSEG	BASE	Segment for initialization data (**sreg** variable with initial value)
@@RSINS1	CSEG	BASE	Segment for initialization data (**sreg1** variable with initial value)
@@INIT	DSEG		Segment for data area (with initial value)
@@DATA	DSEG		Segment for data area (without initial value)
@@INIS	DSEG	SADDR2	Segment for data area (**sreg** variable with initial value)
@@DATS	DSEG	SADDR2	Segment for data area (**sreg** variable without initial value)
@@INIS1	DSEG	SADDR	Segment for data area (**sreg1** variable with initial value)
@@DATS1	DSEG	SADDR	Segment for data area (**sreg1** variable without initial value)
@@BITS	BSEG	SADDR2	Segment for **boolean** type and **bit** type variables
@@BITS1	BSEG	SADDR	Segment for _ _**boolean 1** type variable
@EXT00	CSEG	AT04080H	Segment for the flash area branch table (only when **-ZF** is specified)[Note]

Note When **-ZF** is specified, the second "@" from the top is changed to "E" in the section name. For details, refer to **B.1.2 Flash memory area** (**@@INIS→@EINIS**, etc.).
Also, it is possible to change the address of the relocation attribute using **#pragma ext_table**.

Remark For **@@VECTnn**, **nn** is determined when the interrupt source is specified by **#pragma vect** (**interrupt**) (**nn**: Number of interrupt vector address).

(2) With large model (when –ML is specified)

Section Name	Segment Type	Relocation Attribute	Description
@@BASE	CSEG	BASE	Segment for **callt** function and **interrupt** function
@@VECTnn	CSEG	AT nnH	Segment for interrupt vector table
@@CODE	CSEG		Segment for ordinary function codes
@@CNST	CSEG		Segment for **const** variables
@@CALF	CSEG	FIXED	Segment for **callf** function
@@CALT	CSEG	CALLT0	Segment for table for **callt** function
@@R_INIT	CSEG		Segment for initialization data (with initial value)
@@R_INIS	CSEG		Segment for initialization data (**sreg** variable with initial value)
@@R_INS1	CSEG		Segment for initialization data (**sreg1** variable with initial value)
@@INIT	DSEG		Segment for data area (with initial value)
@@DATA	DSEG		Segment for data area (without initial value)
@@INIS	DSEG	SADDR2	Segment for data area (**sreg** variable with initial value)
@@DATS	DSEG	SADDR2	Segment for data area (**sreg** variable without initial value)
@@INIS1	DSEG	SADDR	Segment for data area (**sreg1** with initial value)
@@DATS1	DSEG	SADDR	Segment for data area (**sreg1** variable without initial value)
@@BITS	BSEG	SADDR2	Segment for **boolean** type and **bit** type variables
@@BITS1	BSEG	SADDR	Segment for _ _**boolean1** type variable
@EXT00	CSEG	AT04080H	Segment for the flash area branch table (only when **-ZF** is specified)[Note]

Note When **-ZF** is specified, the second "@" from the top is changed to "E" in the section name. For details, refer to **B.1.2 Flash memory area (@@INIS→@EINIS**, etc.).
Also, it is possible to change the address of the relocation attribute using **#pragma ext_table**.

Remark For the **@@VECTnn**, **nn** is determined when the interrupt source is specified by **#pragma vect** (**interrupt**) (**nn**: Number of interrupt vector address).

(3) With medium model and location 00H (when –MM and –CS0 are specified)

Section Name	Segment Type	Relocation Attribute	Description
@@BASE	CSEG	BASE	Segment for **callt** function and **interrupt** function
@@VECTnn	CSEG	AT nnH	Segment for interrupt vector table
@@CODE	CSEG		Segment for ordinary function codes
@@CNSTS	CSEG	BASE	Segment for const variables
@@CALF	CSEG	FIXED	Segment for **callf** function
@@CALT	CSEG	CALLT0	Segment for table for **callt** function
@@R_INIT	CSEG		Segment for initialization data (with initial value)
@@R_INIS	CSEG		Segment for initialization data (**sreg** variable with initial value)
@@R_INS1	CSEG		Segment for initialization data (**sreg1** variable with initial value)
@@INIT	DSEG		Segment for data area (with initial value)
@@DATA	DSEG		Segment for data area (without initial value)
@@INIS	DSEG	SADDR2	Segment for data area (**sreg** variable with initial value)
@@DATS	DSEG	SADDR2	Segment for data area (**sreg** variable without initial value)
@@INIS1	DSEG	SADDR	Segment for data area (**sreg1** variable with initial value)
@@DATS1	DSEG	SADDR	Segment for data area (**sreg1** variable without initial value)
@@BITS	BSEG	SADDR2	Segment for **boolean** type and **bit** type variables
@@BITS1	BSEG	SADDR	Segment for _ _**boolean1** type variable
@EXT00	CSEG	AT04080H	Segment for the flash area branch table (only when **-ZF** is specified)[Note]

Note When **-ZF** is specified, the second "@" from the top is changed to "E" in the section name. For details, refer to **B.1.2 Flash memory area** (**@@INIS→@EINIS**, etc.).
Also, it is possible to change the address of the relocation attribute using **#pragma ext_table**.

Remark For the **@@VECTnn**, **nn** is determined when the interrupt source is specified by **#pragma vect** (**interrupt**) (**nn**: Number of interrupt vector address).

(4) With medium model and location 0FH (when –MM and –CS15 are specified)

Section Name	Segment Type	Relocation Attribute	Description
@@BASE	CSEG	BASE	Segment for **callt** function and **interrupt** function
@@VECTnn	CSEG	AT nnH	Segment for interrupt vector table
@@CODE	CSEG		Segment for ordinary function codes
@@CNSTM	CSEG	PAGE64K	Segment for **const** variables
@@CALF	CSEG	FIXED	Segment for **callf** function
@@CALT	CSEG	CALLT0	Segment for table for **callt** function
@@R_INIT	CSEG		Segment for initialization data (with initial value)
@@R_INIS	CSEG		Segment for initialization data (**sreg** variable with initial value)
@@R_INS1	CSEG		Segment for initialization data (**sreg1** variable with initial value)
@@INITM	DSEG	PAGE64K	Segment for data area (with initial value)
@@DATAM	DSEG	PAGE64K	Segment for data area (without initial value)
@@INIS	DSEG	SADDR2	Segment for data area (**sreg** variable with initial value)
@@DATS	DSEG	SADDR2	Segment for data area (**sreg** variable without initial value)
@@INIS1	DSEG	SADDR	Segment for data area (**sreg1** variable with initial value)
@@DATS1	DSEG	SADDR	Segment for data area (**sreg1** variable without initial value)
@@BITS	BSEG	SADDR2	Segment for **boolean** type and **bit** type variables
@@BITS1	BSEG	SADDR	Segment for _ _**boolean1** type variable
@EXT00	CSEG	AT04080H	Segment for the flash area branch table (only when **-ZF** is specified)[Note]

Note When **-ZF** is specified, the second "@" from the top is changed to "E" in the section name. For details, refer to **B.1.2 Flash memory area** (**@@INIS**→**@EINIS**, etc.).
Also, it is possible to change the address of the relocation attribute using **#pragma ext_table**.

Remark For the **@@VECTnn**, **nn** is determined when the interrupt source is specified by **#pragma vect** (**interrupt**) (**nn**: Number of interrupt vector address).

B.1.2 Flash memory area

(1) With small model (when –MS is specified)

Section Name	Segment Type	Relocation Attribute	Description
@ECODES	CSEG	BASE	Segment for normal function codes
@ECNSTS	CSEG	BASE	Segment for **const** variables
@ERSINIT	CSEG	BASE	Segment for initialization data (with initial value)
@ERSINIS	CSEG	BASE	Segment for initialization data (**sreg** variable with initial value)
@ERSINS1	CSEG	BASE	Segment for initialization data (**sreg1** variable with initial value)
@EINIT	DSEG		Segment for data area (with initial value)
@EDATA	DSEG		Segment for data area (without initial value)
@EINIS	DSEG	SADDR2	Segment for data area (**sreg** variable with initial value)
@EDATS	DSEG	SADDR2	Segment for data area (**sreg** variable without initial value)
@EINIS1	DSEG	SADDR	Segment for data area (**sreg1** variable with initial value)
@EDATS1	DSEG	SADDR	Segment for data area (**sreg1** variable without initial value)
@EBITS	BSEG	SADDR2	Segment for **boolean** type and **bit** type variables
@EBITS1	BSEG	SADDR	Segment for _ _**boolean 1** type variable

(2) With large model (when –ML is specified without 2-byte alignment)

Section Name	Segment Type	Relocation Attribute	Description
@ECODE	CSEG		Segment for normal function codes
@ECNST	CSEG		Segment for **const** variables
@ER_INIT	CSEG		Segment for initialization data (with initial value)
@ER_INIS	CSEG		Segment for initialization data (**sreg** variable with initial value)
@ER_INS1	CSEG		Segment for initialization data (**sreg1** variable with initial value)
@EINIT	DSEG		Segment for data area (with initial value)
@EDATA	DSEG		Segment for data area (without initial value)
@EINIS	DSEG	SADDR2	Segment for data area (**sreg** variable with initial value)
@EDATS	DSEG	SADDR2	Segment for data area (**sreg** variable without initial value)
@EINIS1	DSEG	SADDR	Segment for data area (**sreg1** with initial value)
@EDATS1	DSEG	SADDR	Segment for data area (**sreg1** variable without initial value)
@EBITS	BSEG	SADDR2	Segment for **boolean** type and **bit** type variables
@EBITS1	BSEG	SADDR	Segment for _ _**boolean1** type variable

(3) With large model (when –ML is specified with 2-byte alignment)

Section Name	Segment Type	Relocation Attribute	Description
@ECODE	CSEG		Segment for normal function codes
@ECNST	CSEG	UNITP	Segment for **const** variables
@ER_INIT	CSEG	UNITP	Segment for initialization data (with initial value)
@ER_INIS	CSEG		Segment for initialization data (**sreg** variable with initial value)
@ER_INS1	CSEG		Segment for initialization data (**sreg1** variable with initial value)
@EINIT	DSEG	UNITP	Segment for data area (with initial value)
@EDATA	DSEG	UNITP	Segment for data area (without initial value)
@EINIS	DSEG	SADDR2	Segment for data area (**sreg** variable with initial value)
@EDATS	DSEG	SADDR2	Segment for data area (**sreg** variable without initial value)
@EINIS1	DSEG	SADDR	Segment for data area (**sreg1** with initial value)
@EDATS1	DSEG	SADDR	Segment for data area (**sreg1** variable without initial value)
@EBITS	BSEG	SADDR2	Segment for **boolean** type and **bit** type variables
@EBITS1	BSEG	SADDR	Segment for _ _**boolean1** type variable

(4) With medium model and location 00H (when –MM and –CS0 are specified)

Section Name	Segment Type	Relocation Attribute	Description
@ECODE	CSEG		Segment for normal function codes
@ECNSTS	CSEG	BASE	Segment for const variables
@ER_INIT	CSEG		Segment for initialization data (with initial value)
@ER_INIS	CSEG		Segment for initialization data (**sreg** variable with initial value)
@ER_INS1	CSEG		Segment for initialization data (**sreg1** variable with initial value)
@EINIT	DSEG		Segment for data area (variable with initial value)
@EDATA	DSEG		Segment for data area (without initial value)
@EINIS	DSEG	SADDR2	Segment for data area (**sreg** variable with initial value)
@EDATS	DSEG	SADDR2	Segment for data area (**sreg** variable without initial value)
@EINIS1	DSEG	SADDR	Segment for data area (**sreg1** variable with initial value)
@EDATS1	DSEG	SADDR	Segment for data area (**sreg1** variable without initial value)
@EBITS	BSEG	SADDR2	Segment for **boolean** type and **bit** type variables
@EBITS1	BSEG	SADDR	Segment for _ _**boolean1** type variable

(5) With medium model and location 0FH (when –MM and –CS15 are specified)

Section Name	Segment Type	Relocation Attribute	Description
@ECODE	CSEG		Segment for normal function codes
@ECNSTM	CSEG	PAGE64K	Segment for **const** variables
@ER_INIT	CSEG		Segment for initialization data (with initial value)
@ER_INIS	CSEG		Segment for initialization data (**sreg** variable with initial value)
@ER_INS1	CSEG		Segment for initialization data (**sreg1** variable with initial value)
@EINITM	DSEG	PAGE64K	Segment for data area (with initial value)
@EDATAM	DSEG	PAGE64K	Segment for data area (without initial value)
@EINIS	DSEG	SADDR2	Segment for data area (**sreg** variable with initial value)
@EDATS	DSEG	SADDR2	Segment for data area (**sreg** variable without initial value)
@EINIS1	DSEG	SADDR	Segment for data area (**sreg1** variable with initial value)
@EDATS1	DSEG	SADDR	Segment for data area (**sreg1** variable without initial value)
@EBITS	BSEG	SADDR2	Segment for **boolean** type and **bit** type variables
@EBITS1	BSEG	SADDR	Segment for _ _**boolean1** type variable

B.2 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 inter(void);                        /* interrupt function prototype declaration */
const int i_cnst = 1;                    /* const variable */
callt void f_clt(void);                  /* callt function prototype declaration */
callf void f_clf(void);                  /* callf 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()                              /* function definition */
{
   int i;
   i = 100;
}

void inter()                             /* interrupt function definition */
{
   unsigned char uc = 0;
   uc++;
   if(b_bit)
   b_bit = 0;
}

callt void f_clt()                       /* callt function definition */
{
}

callf void f_clf()                       /* callf 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 RA78K4 Online Help for details.

```
; 78K/IV Series C Compiler V2.30 Assembler Source
;                                        Date:XX XXX XXXX Time:xx:xx:xx

; Command   : -c4026 sample.c -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$CHGSFR(15)
$PROCESSOR(4026)
$NODEBUG
$NODEBUGA
$KANJICODE SJIS
$TOL_INF       03FH, 0230H, 00H, 08021H, 00H

       PUBLIC _inter
       PUBLIC _i_cnst
       PUBLIC ?f_clt
       PUBLIC _f_clf
       PUBLIC _b_bit
       PUBLIC _l_init
       PUBLIC _i_data
       PUBLIC _sr_inis
       PUBLIC _sr_dats
       PUBLIC _main
       PUBLIC _f_clt
       PUBLIC _@vect06

@@BITS  BSEG    SADDR2                          ; Segment for boolean type variable
_b_bit  DBIT

@@CNST  CSEG                                    ; Segment for const variable
_i_cnst:        DW      01H     ; 1

@@R_INIT        CSEG                            ; Segment for initialization data (external variable
       DW       00002H,00000H  ; 2               with initial value)

@@INIT  DSEG                                    ; Segment for data area (external variable with initial
_l_init:        DS      (4)                      value)
```

```
@@DATA  DSEG                                    ; Segment for data area (external variable without
_i_data:        DS      (2)                       initial value)

@@R_INIS        CSEG                            ; Segment for initialization data (sreg variable with
        DW      03H     ; 3                       initial value)

@@INIS  DSEG    SADDR2                          ; Segment for data area (sreg variable with initial
_sr_inis:       DS      (2)                       value)

@@DATS  DSEG    SADDR2                          ; Segment for data area (sreg variable without initial
_sr_dats:       DS      (2)                       value)

@@CALT  CSEG    CALLT0                          ; Segment for callt function
?f_clt: DW      _f_clt

; line     1 : #pragma INTERRUPT INTP0 inter rb1      /* interrupt vector */
; line     2 :
; line     3 : void inter(void);                     /* interrupt 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 : callf void f_clf(void);               /* callf function prototype declaration */
; line     7 : boolean b_bit;                        /* boolean type variable */
; line     8 : long l_init = 2;                      /* external variable with initial value */
; line     9 : int i_data;                           /* external variable without initial value */
; line    10 : sreg int sr_inis = 3;                 /* sreg variable with initial value */
; line    11 : sreg int sr_dats;                     /* sreg variable without initial value */
; line    12 :
; line    13 : void main()                          /* function definition */
; line    14 : {

@@CODE  CSEG                                    ; Segment for code portion
_main:
        push    rp3
; line    15 :     int i;
; line    16 :     i = 100;
        movw    rp3,#064H       ; 100
; line    17 : }
        pop     rp3
        ret
; line    18 :
; line    19 : void inter()                         /* interrupt function definition */
; line    20 : {

@@BASE  CSEG    BASE                            ; Segment for callf/interrupt function
_inter:
       sel     RB1
```

```
        push    rp3
; line    21 :     unsigned char uc = 0;
        mov     r6,#00H ; 0
; line    22 :     uc++;
        inc     r6
; line    23 :     if(b_bit)
        bf      _b_bit,$L0005
; line    24 :         b_bit = 0;
        clr1    _b_bit
L0005:
; line    25 : }
        pop     rp3
        reti
; line    26 :
; line    27 : callt void f_clt()              /* callt function definition */
; line    28 : {
_f_clt:
; line    29 : }
        ret
; line    30 :
; line    31 : callf void f_clf()              /* callf function definition */
; line    32 : {

@@CALF  CSEG    FIXED                          ; Segment for callf function
_f_clf:
; line    33 : }
        ret

@@VECT06        CSEG    AT    0006H            ; Segment for interrupt vector table
_@vect06:
        DW      _inter
        END


; Target chip : uPD784026
; Device file : Vx.xx
```

APPENDIX C LIST OF RUNTIME LIBRARIES

Table C-1 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** and **cstarte** are called in the format with _@ attached to the top.

All runtime libraries except **hdwinit** and **boot_main** are supported when the **-ZF** option is specified.

No library support is available for operations not in Table C-1. The compiler executes inline expansion.

long addition and subtraction, **and/or/xor** and shift may be expanded inline.

Table C-1. List of Runtime Libraries (1/5)

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 **float**.
Complement	lscom	Obtains one's complement of **signed long**.
	lucom	Obtains one's complement of **unsigned long**.
NOT	lsnot	Negates **signed long**.
	lunot	Negates **unsigned long**.
	fnot	Negates **float**.
Multiply	lsmul	Performs multiplication between two **signed long** data.
	lumul	Performs multiplication between two **unsigned long** data.
	fmul	Performs multiplication between two **float** data.
Divide	csdiv	Performs division between two **signed char** data.
	isdiv	Performs division between two **signed int** data.
	lsdiv	Performs division between two **signed long** data.
	ludiv	Performs division between two **unsigned long** data.
	fdiv	Performs division between two **float** data.
Remainder	csrem	Obtains remainder after division between two **signed char** data.
	isrem	Obtains remainder after division between two **signed int** data.
	lsrem	Obtains remainder after division between two **signed long** data.
	lurem	Obtains remainder after division between two **unsigned long** data.

Table C-1. List of Runtime Libraries (2/5)

Classification	Function Name	Function
Add	lsadd	Performs addition between two **signed long** data.
	luadd	Performs addition between two **unsigned long** data.
	fadd	Performs addition between two **float** data.
Subtract	lssub	Performs subtraction between two **signed long** data.
	lusub	Performs subtraction between two **unsigned long** data.
	fsub	Performs subtraction between two **float** data.
Shift Left	lslsh	Shifts **signed long** to the left.
	lulsh	Shifts **unsigned long** to the left.
Shift Right	lsrsh	Shifts **signed long** to the right.
	lursh	Shifts **unsigned long** to the right.
Compare	lscmp	Compares two **signed long** data.
	lucmp	Compares two **unsigned long** data.
	fcmp	Compares two **float** data.
Bitwise AND	lsband	Performs bitwise AND operation between two **signed long** data.
	luband	Performs bitwise AND operation between two **unsigned long** data.
Bitwise OR	lsbor	Performs bitwise OR operation between two **signed long** data.
	lubor	Performs bitwise OR operation between two **unsigned long** data.
Bitwise XOR	lsbxor	Performs bitwise XOR operation between two **signed long** data.
	lubxor	Performs bitwise XOR operation between two **unsigned long** data.
Logical AND	fand	Performs logical AND operation between two **float** data.
Logical OR	for	Performs logical OR operation between two **float** data.
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**.
Type conversion from bit	btol	Converts bit to **long**.
Preprocess/ postprocess	hdwinit	Initializes peripheral units (**sfr**) immediately after CPU has been reset.

Table C-1. List of Runtime Libraries (3/5)

Classification	Function Name	Function
Startup routine	cstart	Startup module (including the startup module for booting) ● **In the case of a startup module for booting**, library.inc, in which a library name **EXTERN** declaration is described in the comments is included. If the library name's **EXTERN** declaration comment is removed, it is used in the flash area. The library can be used in the boot area. **EXTERN** declarations **_@vect00** to **@vect3e** are executed and are located in the flash area. Set an interrupt vector table for interrupt functions. Secure an area (2 x 32 bytes, 3 x 32 bytes for the medium model and large model) to register functions by the **atexit** function, and let the top label name be **_@FNCTBL**. Secure a break area (32 bytes, 64 bytes in the large model) and let the top label name be **_@MEMTOP**, then let the area's next address label name be **_@MEMBTM**. Define the reset vector table's segment as follows and specify the top address of the startup module. **@@VECT00 CSEG AT 0000H** **DW _@cstart** Specify **LOCATION**. Set the **V, U, T** and **W** registers to 0 (small model only). Set the **V, U, T** and **W** registers to 0 (**LOCATION 0**) and 0FH (**LOCATION 15**) (medium model only). Set the register bank to **RB0**. Set variable **_errno** input in the error No to 0. Set the variable **_@FNCENT** which inputs the number of functions registered by the **atexit** function to. Set the address of **_@MEMTOP** in variable **_@BRAKADR** as the initial break value. Set 1 as the initial value in the variable **_@SEED** which is the source of pseudo random numbers for the **rand** function. Execute 0 clearing of data from initialization data copy processing and external data without initialization values.

Table C-1. List of Runtime Libraries (4/5)

Classification	Function Name	Function
Startup routine	cstart	Startup module (including startup modules for booting) ● **In the case of a startup module for booting (for flash)** Call the **boot_main** function (user program). Branch to the flash area's branch table top (**ITBLTOP**) and move processing to the startup module for flash memory. Declare the following labels and variables (distinguish between upper case and lower case letters). The user is prohibited to define these symbols. **_@FNCTBL** (3 bytes: Medium model, large model) **_@MEMTOP** (3 bytes: Large model) **_@MEMBTM** (3 bytes: Large model) **_errno** (2 bytes) **_@FNCENT** (2 bytes) **_@BRKADR** (2 bytes/3 bytes: Large model) **_@SEED** (4 bytes) _@DIVR (4 bytes) **_@LDIVR** (8 bytes) **_@TOKPRT** (2 bytes/3 bytes: Large model)
		● **In the case of a startup module for booting** Call the **main** function (user program). Call the **exit** function by parameter 0.
	cstarte	Startup module for flash memory Define the flash area branch table for branching to the startup module for flash memory (**ITBLTOP** is the top address for the flash area branch table). **@EVECT00 CSEG AT ITBLTOP** **BR _@cstarte** Set the final address of the stack area + 1 in the stack pointer (SP). Execute 0 clearing of data from initialization data copy processing and external data without initialization values. Call the **main** function. Call the **exit** function by parameter 0.
Flash compatibility	boot_main	Execute boot area **main** function processing (function prototype: **void boot_main (void)**;). This function returns without doing anything. However, as necessary, the user, by creating it, can execute processing which suit's the user's purpose. **Example**: In cases where update processing of the flash area program is executed by referring to SFR, etc.
	vect00 to 3e	Create an interrupt vector table when the **-ZF** option is specified (function prototype: **void vect00(void)**;, ..., **void vect3e (void)**). Specify the top address value of the interrupt function located in the flash area in the interrupt vector table.

Table C-1. List of Runtime Libraries (5/5)

Classification	Function Name	Function
Auxiliary	addwc	For replacing the fixed-type instruction pattern
	anda0	
	aX3de	
	aX3whl	
	aXxwhl	
	clrhw	
	cmpa0	
	cmpax0	
	cmpaxf	
	cmpbc0	
	cmpbcf	
	eX2de	
	eX4de	
	mova0	
	movax1	
	movaxs	
	movbcf	
	movdes	
	movs0	
	movsax	
	muluwt	
	muluww	
	mulwde	
	mulwhl	
	sladd	
	slsdiv	
	slsmul	
	slsrem	
	slsub	
	sludiv	
	slumul	
	slurem	
	swtbla	Converts **switch** branch table to 2-byte table.

APPENDIX D LIST OF LIBRARY STACK CONSUMPTION

Table D-1 shows the number of stacks consumed from the standard libraries.

Table D-1. List of Standard Library Stack Consumption (1/4)

Classification	Function Name	Small Model	Medium Model	Large Model
ctype.h	isalnum	0	0	0
	salpha	0	0	0
	iscntrl	0	0	0
	isdigit	0	0	0
	isgraph	0	0	0
	islower	0	0	0
	isprint	0	0	0
	ispunct	0	0	0
	isspace	0	0	0
	isupper	0	0	0
	isxdigit	0	0	0
	tolower	0	0	0
	toupper	0	0	0
	isascii	0	0	0
	toascii	0	0	0
	_tolower	0	0	0
	_toupper	0	0	0
	tolow	0	0	0
	toup	0	0	0
setjmp.h	setjmp	6	6	0
	longjmp	0	0	0
stdarg.h	va_arg	0	0	0
	va_start	0	0	0
	va_end	0	0	0
stdio.h	sprintf	56 (115)	56 (116)	55 (119)[Note]
	sscanf	293 (334)	293 (335)	293 (341)[Note]
	printf	65 (116)	67 (118)	71 (121)[Note]
	scanf	304 (336)	308 (338)	308 (344)[Note]
	vprintf	65 (116)	67 (118)	71 (121)[Note]
	vsprintf	56 (115)	56 (116)	55 (119)[Note]
	getchar	0	0	0
	gets	7	7	9
	putchar	0	0	0
	puts	5	5	6

Note Values in parentheses are for when the version that supports floating-point numbers is used.

Table D-1. List of Standard Library Stack Consumption (2/4)

Classification	Function Name	Small Model	Medium Model	Large Model
stdlib.h	atoi	11	11	1
	atol	11	11	1
	strtol	14	17	21
	strtoul	14	17	21
	calloc	11	11	18
	free	9	9	12
	malloc	9	9	12
	realloc	14	14	20
	abort	0	0	0
	atexit	0	0	3
	exit	n+3	n+3	n+3[Note 1]
	abs	0	0	0
	div	6	6	6
	labs	0	0	0
	ldiv	8	8	11
	brk	3	3	6
	sbrk	3	3	6
	atof	39	39	40
	strtod	39	39	40
	itoa	6	6	8
	ltoa	10	10	12
	ultoa	10	10	11
	rand	5	5	5
	srand	0	0	0
	bsearch	25+n	26+n	29+n[Note 2]
	qsort	36+n	43+n	44+n[Note 3]
	strbrk	3	3	6
	strsbrk	3	3	6
	stritoa	6	6	8
	strltoa	10	10	13
	strultoa	10	10	11

Notes 1. n is the total stack consumption among external functions registered by the **atexit** function.

2. n is the stack consumption of external functions called from **bsearch**.

3. n is (20 + stack consumption of external functions called from **qsort**) × (1 + number of times recursive calls occurred).

Table D-1. List of Standard Library Stack Consumption (3/4)

Classification	Function Name	Small Model	Medium Model	Large Model
string.h	memcpy	0	0	3
	memmove	0	0	6
	strcpy	0	0	3
	strncpy	0	0	3
	strcat	0	0	3
	strncat	0	0	3
	memcmp	0	0	0
	strcmp	0	0	0
	strncmp	0	0	0
	memchr	0	0	0
	strchr	0	0	0
	strcspn	0	0	3
	strpbrk	0	0	3
	strrchr	0	0	0
	strspn	0	0	3
	strstr	2	2	3
	strtok	0	0	6
	memset	0	0	0
	strerror	3	6	6
	strlen	0	0	0
	strcoll	0	0	0
	strxfrm	2	2	3
math.h	acos	31	31	31
	asin	31	31	31
	atan	28	28	28
	atan2	28	28	28
	cos	26	26	26
	sin	26	26	26
	tan	33	33	33
	cosh	31	31	31
	sinh	31	31	31
	tanh	37	37	37
	exp	28	28	28
	frexp	0 (14)	0 (14)	0 (15)[Note]
	ldexp	0 (11)	0 (11)	0 (12)[Note]
	log	30	30	30
	log10	30	30	30
	modf	7 (11)	7 (11)	7 (12)[Note]

Note Values in parentheses are for when an operation exception occurs.

Table D-1. List of Standard Library Stack Consumption (4/4)

Classification	Function Name	Small Model	Medium Model	Large Model
math.h	pow	30	30	30
	sqrt	12	12	12
	ceil	7 (11)	7 (11)	7 (12)[Note 1]
	fabs	0	0	0
	floor	7 (11)	7 (11)	7 (12)[Note 1]
	fmod	6 (11)	6 (11)	6 (12)[Note 1]
	matherr	0	0	0
	asinf	31	31	31
	atanf	28	28	28
	atan2f	28	28	28
	cosf	26	26	26
	sinf	26	26	26
	tanf	33	33	33
	coshf	31	31	31
	sinhf	31	31	31
	tanhf	37	37	37
	expf	28	28	28
	rexpf	0 (14)	0 (14)	0 (15)[Note 1]
	ldexpf	0 (11)	0 (11)	0 (12)[Note 1]
	logf	30	30	30
	log10f	30	30	30
	modff	7 (11)	7 (11)	7 (12)[Note 1]
	powf	30	30	30
	sqrtf	12	12	12
	ceilf	7 (11)	7 (11)	7 (12)[Note 1]
	fabsf	0	0	0
	floorf	7 (11)	7 (11)	7 (12)[Note 1]
	fmodf	6 (11)	6 (11)	6 (12)[Note 1]
assert.h	_ _assertfail	76 (127)	78 (129)	85 (135)[Note 2]

Notes 1. Values in parentheses are for when an operation exception occurs.

2. Values in parentheses are for when the **printf** version that supports floating-point numbers is used.

Table D-2 shows the number of stacks consumed from the runtime libraries.

Table D-2. List of Runtime Library Stack Consumption (1/3)

Classification	Function Name	Small Model	Medium Model	Large Model
Increment	lsinc	0	0	0
	luinc	0	0	0
	finc	15 (24)	15 (24)	16 (26)[Note]
Decrement	lsdec	0	0	0
	ludec	0	0	0
	fdec	15 (24)	15 (24)	16 (26)[Note]
Sign reverse	lsrev	2	2	2
	lurev	2	2	2
	frev	0	0	0
1's complement	lscom	0	0	0
	lucom	0	0	0
Logical NOT	lsnot	0	0	0
	lunot	0	0	0
	fnot	0	0	0
Multiply	lsmul	2	2	2
	lumul	2	2	2
	fmul	8 (17)	8 (17)	9 (19)[Note]
Divide	csdiv	4	4	4
	isdiv	6	6	6
	lsdiv	13	13	13
	ludiv	6	6	6
	fdiv	8 (17)	8 (17)	9 (19)[Note]
Remainder	csrem	4	4	4
	isrem	6	6	6
	lsrem	13	13	13
	lurem	6	6	6
Add	lsadd	0	0	0
	luadd	0	0	0
	fadd	8 (17)	8 (17)	9 (19)[Note]
Subtract	lssub	0	0	0
	lusub	0	0	0
	fsub	8 (17)	8 (17)	9 (19)[Note]

Note Values in parentheses are for when an operation exception occurs (when the **matherr** function included with the compiler is used).

Table D-2. List of Runtime Library Stack Consumption (2/3)

Classification	Function Name	Small Model	Medium Model	Large Model
Shift left	lslsh	0	0	0
	lulsh	0	0	0
Shift right	lsrsh	0	0	0
	lursh	0	0	0
Compare	lscmp	0	0	0
	lucmp	0	0	0
	fcmp	2 (17)	2 (17)	2 (19)[Note]
Bit AND	lsband	0	0	0
	luband	0	0	0
Bit OR	lsbor	0	0	0
	lubor	0	0	0
Bit XOR	lsbxor	0	0	0
	lubxor	0	0	0
Logical AND	fand	0	0	0
Logical OR	for	0	0	0
Conversion from floating-point number	ftols	2	2	2
	ftolu	2	2	2
Conversion to floating-point number	lstof	2	2	2
	lutof	2	2	2
Conversion from bit	btol	2	2	2
Startup routine	cstart	3	3	3

Note Values in parentheses are for when an operation exception occurs (when the **matherr** function included with the compiler is used).

Table D-2. List of Runtime Library Stack Consumption (3/3)

Classification	Function Name	Small Model	Medium Model	Large Model
Auxiliary	addwc	0	5	5
	anda0	0	0	0
	aX3de	0	0	0
	aX3whl	0	0	0
	aXxwhl	0	0	0
	clrhw	0	0	0
	cmpa0	0	0	0
	cmpax0	0	0	0
	cmpaxf	0	0	0
	cmpbc0	0	0	0
	cmpbcf	0	0	0
	eX2de	0	0	0
	eX4de	0	0	0
	mova0	0	0	0
	movax1	0	0	0
	movaxs	2	3	5
	movbcf	0	0	0
	movdes	4	5	5
	movs0	4	7	5
	movsax	4	7	5
	muluwt	—	—	0
	muluww	—	—	0
	mulwde	—	—	0
	mulwhl	—	—	0
	sladd	3	3	3[Note]
	slsdiv	3	3	3[Note]
	slsmul	9	9	9[Note]
	slsrem	25	25	25[Note]
	slsub	5	5	5[Note]
	sludiv	9	9	9[Note]
	slumul	9	9	9[Note]
	slurem	13	11	11[Note]
	swtbla	—	—	0[Note]

Note Stack correction for the 4 bytes used for placing an argument when a function is called is performed on the side of called function.

APPENDIX E INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

[MEMO]

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America
NEC Electronics Inc.
Corporate Communications Dept.
Fax: +1-800-729-9288
+1-408-588-6130

Europe
NEC Electronics (Europe) GmbH
Market Communication Dept.
Fax: +49-211-6503-274

South America
NEC do Brasil S.A.
Fax: +55-11-6462-6829

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: +82-2-528-4411

Taiwan
NEC Electronics Taiwan Ltd.
Fax: +886-2-2719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: +81- 44-435-9608

I would like to report the following error/make the following suggestion:

Document title: _____

Document number: _____ Page number: _____

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	❑	❑	❑	❑
Technical Accuracy	❑	❑	❑	❑
Organization	❑	❑	❑	❑