User's Manual

NEC

CC78K0 Ver. 3.60

C Compiler

Language

Target Device
78K0 Series

Document No. U17016EJ1V0UM00 (1st edition)
Date Published April 2004 CP(K)

Printed in Japan

[MEMO]

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC Electronics product in your application, please contact the NEC Electronics 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.

[GLOBAL SUPPORT]
http://www.necel.com/en/support/support.html

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

NEC Electronics (Europe) GmbH
Duesseldorf, Germany
Tel: 0211-65030

- **Sucursal en España**
  Madrid, Spain
  Tel: 091-504 27 87
- **Succursale Française**
  Vélizy-Villacoublay, France
  Tel: 01-30-67 58 00
- **Filiale Italiana**
  Milano, Italy
  Tel: 02-66 75 41
- **Branch The Netherlands**
  Eindhoven, The Netherlands
  Tel: 040-244 58 45
- **Tyskland Filial**
  Taeby, Sweden
  Tel: 08-63 80 820
- **United Kingdom Branch**
  Milton Keynes, UK
  Tel: 01908-691-133

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-558-3737

NEC Electronics Shanghai Ltd.
Shanghai, P.R. China
Tel: 021-5888-5400

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

NEC Electronics Singapore Pte. Ltd.
Novena Square, Singapore
Tel: 6253-8311

J04.1

[MEMO]

INTRODUCTION

The **CC78K0 C Compiler** (hereinafter referred to as this C compiler) was developed based on **CHAPTER 2 ENVIRONMENT** and **CHAPTER 3 LANGUAGE** in the **Draft Proposed American National Standard for Information Systems — Programming Language C** (December 7, 1988). Therefore, by compiling C source programs conforming to the ANSI standard with this C compiler, applied products for the 78K0 Series can be developed.

The **CC78K0 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 **CC78K0 C Compiler Operation (U17017E)**.

For the architecture of 78K0 Series, refer to the user's manual of each product of 78K0 Series.

[Target Devices]

Software for the 78K0 Series microcontrollers can be developed with this C compiler.
Note that an optional device file corresponding to a target device is necessary.

[Readers]

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

[Organization]

This manual consists of the following 13 chapters and appendixes:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[How to Read This Manual]

- For those who are not familiar with C compilers or C language:
  Read from **Chapter 1**, as this manual covers from the program control structures of C to the extended functions of this C compiler. In **Chapter 1**, an example of C source program is used to show the reference part in this manual.
- For those who are familiar with C compilers or C language:
  The language specifications of this C compiler conform to the **ANSI Standard C**. Therefore, you may start from **Chapter 11** that explains the extended functions unique to this C compiler. When reading Chapter 11, also refer to the user's manual supplied with the target device in the 78K0 Series if necessary.

[Related Documents]

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

Documents related to development tools (user's manuals)

Document Name		Document No.
CC78K0 Ver. 3.60 C Compiler	Operation	U17017E
	Language	This document
RA78K0 Ver. 3.70 Assembler Package	Operation	U17015E
	Language	U17014E
	Structured assembly language	U11789E
SM78K0 Series Ver. 2.52 System Simulator	Operation	U16768E
ID78K0-NS Ver. 2.52 Integrated Debugger	Operation	U16488E
PM plus Ver. 5.20		U16934E

[Reference]

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

[Terms]

RTOS = **78K0 Series Real-time OS RX78K0**

[Conventions]

The following symbols and abbreviations are used in this manual:

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

[MEMO]

CONTENTS

LIST OF FIGURES

LIST OF TABLES (1/2)

LIST OF TABLES (2/2)

[MEMO]

CHAPTER 1 GENERAL

The CC78K0 Series C Compiler is a language processing program which converts a source program written in the C language for the 78K0 Series or ANSI-C into machine language. By the CC78K0 Series C compiler, object files or assembler source files for the 78K0 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 (namely, a programmer in this case) 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 glance and the programmer is likely to make errors in coding.

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

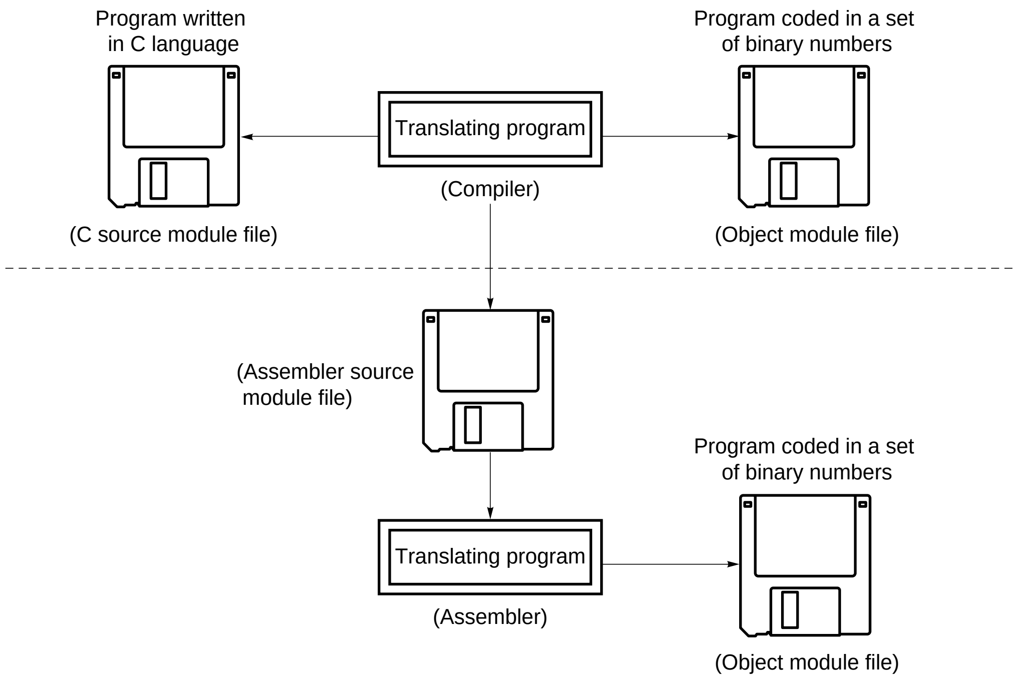
As compared with assembly language programs, it can be said that programs written in C have easy-to-understand logic structure.

C has a rich set of parts called functions for use in creating programs. In other words, the 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 to 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
- RA78K0 assembler package

Assembler..................................	for converting assembly language into machine language
Linker..	for linking object module files
	for determining location address of relocatable segment
Object converter	for conversion to HEX-format object module file
Librarian......................................	for creating library files

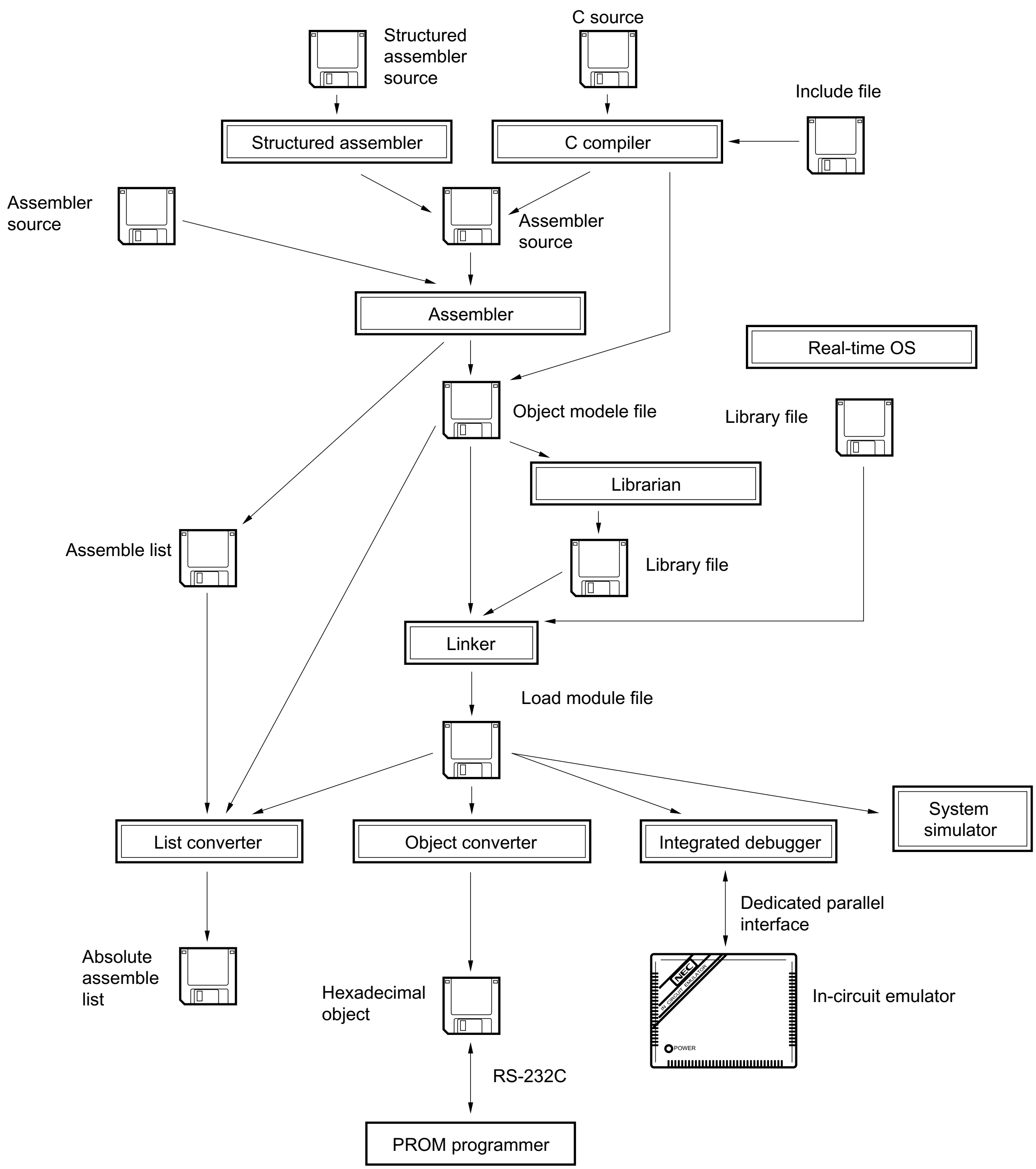
- Debugger (for 78K0) .... for debugging C source module files

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

<1> Divide the product into functions.
<2> Creates a C source module for each function.
<3> Translates each C source module.
<4> Registers the modules to be used frequently in the library.
<5> Links object module files.
<6> Debugs each module.
<7> Converts object modules into HEX-format object files.

As mentioned earlier, 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.

```
Definition of variables/constants  ——  Definition of each data, variable, and macro instruction

main (arguments)                   ——  Header of the function main
{
        statement1;               ┐
        statement2;               │
        function1 (arguments);    ├——  Body of the function main
        function2 (arguments);    ┘

}
function1 (arguments)             ┐
{                                 │
                                  │
        statement1;               ├——  Function 1
        statement2;               │
                                  ┘

}
function2 (arguments)             ┐
{                                 │
        statement1;               ├——  Function 2
        statement2;               │
}                                 ┘
```

An actual C source program looks like this.

```
#define TRUE    1
#define FALSE   0                 #define xxx xxx ....... Preprocessor directive (macro definition)   <6>
#define SIZE    200

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

char mark[SIZE+1];                char xxx ............. Type declarator   <1>   External definition   <5>
main()                            xx [xx]..................................................................Operator   <2>
{
   int i,prime,k,count;           int xxx ........................................................ Type declarator   <1>

   count=0;                       xx = xx..........................................................................Operator   <2>

   for(i=0;i<=SIZE;i++)           for (xx;xx;xx) xxx ;......................................... Control structure   <3>
      mark[i]=TRUE;

   for(i=0;i<=SIZE;i++){
      if(mark[i]){
         prime=i+i+3;             xxx = xxx + xxx + xxx...............................................Operator   <2>
         printf("%6d",prime);     xxx (xxx) ; ..............................................................Operator   <2>

          count++;
         if((count%8)==0) putchar('\n');    if (xxx) xxx ; ...................... Control structure   <3>
         for(k=i+prime;k<=SIZE;k+=prime)
          mark[k]=FALSE;
      }
   }
   printf("\n%d primes found.",count);      xxx (xxx) ;.......................................Operator   <2>
}

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

   j=i;
   ss=s;
}

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

<1> Declaration of type and storage class

The data type and storage class of an identifier that indicates a data object are declared. For details, see **CHAPTER 3 DECLARATION OF TYPES AND STORAGE CLASSES**.

<2> Operator and expression

These are the statements, which instructs the compiler to perform an arithmetic operation, logical operation, assignment, or like. For details, see **CHAPTER 5 OPERATORS AND EXPRESSIONS**.

<3> Control structure

This is a statement that specifies the program flow. C has several instructions for each of control structures such as Conditional control, Iteration, and Branch. For details, see **CHAPTER 6 CONTROL STRUCTURES OF C LANGUAGE**.

<4> Structure or union

A structure or union is declared. A structure is a data object that contains several subobjects or members that may have different types. A union is defined when two or more variables share the same memory. For details, see **CHAPTER 7 STRUCTURES AND UNIONS**.

<5> External definition

A function or external object is declared. A function is 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> Preprocessor directive

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

<7> Declaration of function prototype

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

1.4 Reminders Before Program Development

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

Table 1-1 Maximum Performance Characteristics of This C Compiler (1/2)

No.	Item	Limit Value/Min. Guaranteed Value
1	Nesting level of compound statements, looping statements, or conditional control statements	45 levels
2	Nesting of conditional translations	255 levels
3	Number of arithmetic type, structure type, pointer to qualify union type or incomplete type, array, and function declarator in a declaration (or any combination of these).	12 levels
4	Nesting of parentheses per expression	32 levels
5	Number of characters which have a meaning as a macro name	256 characters
6	Number of characters which 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 which has 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	Nest of function calls	40 levels
20	Number of labels within a function	33 labels

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.

Table 1-1 Maximum Performance Characteristics of This C Compiler (2/2)

No.	Item	Limit Value/Min. Guaranteed Value
21	Total size of code, data, and stack segments per object module	65,535 bytes
22	Number of members per structure or union	256 members
23	Number of **enum** constants per enumeration	255 constants
24	Nest of structures or unions inside a structure or union	15 levels
25	Nest 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	Number of banks that can be used for function definition in one file	1 bank
29	Nest of macros	200
30	Number of -I include file path specifications	64

1.5 Features of This C Compiler

This C compiler has extended functions for CPU code generations that are not supported by the ANSI (American National Standards Institute) Standard C. The extended functions of the C compiler allow the special function registers for the 78K0 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 to help shorten object code and improve execution speed:

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

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

(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 the register or saddr area are shorter in code length than those to memory. This helps shorten object and improves program execution speed as well.

(3) Usage of **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.

(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 post-processing (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 with argument/automatic variables. For the details, refer to **Section 11.5 (5) noauto function**.

(6) **norec**/_ _**leaf** functions

Functions declared as **norec**/_ _**leaf** do not output code for preprocessing and post-processing (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 with argument/automatic variables and is not allowed to call a function. For the details, refer to **Section 11.5 (6) norec function**.

(7) bit type variables and **boolean**/_ _**boolean** type variables

Variables having a 1-bit storage area are generated. By using the **bit** type variable or **boolean**/_ _**boolean** type variable, the **saddr** area can be accessed in bit units.

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

(8) 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.

(9) Interrupt functions

The preprocessor directive outputs a vector table and outputs an object code corresponding to the interrupt. This directive allows programming of interrupt functions in the C source level.

(10) Interrupt function qualifier

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

(11) Interrupt functions

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

(12) CPU control instructions

Each of the following instruction is embedded in the 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

(13) **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.

(14) Absolute address access function

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

(15) 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.

(16) Function to change compiler output section name

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

(17) Binary constant description function

Binary can be described in the C source.

(18) Module name change functions

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

(19) Rotate function

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

(20) 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.

(21) 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.

(22) BCD operation function

This function uses direct inline expansion to output the code that performs a BCD operation on the operation value in an object. A BCD operation is an operation for converting each digit of a decimal number into binary and storing it in 4 bits.

(23) Bank function

The bank function stores the function body in the bank area, and calls the function via a library for bank function call. Configuration of a code that exceeds 64 KB is therefore possible.

(24) Table jump function

The table jump function outputs the code that calls a function corresponding to the value of an operation. Can perform a table jump using a combination of different function attributes. Can perform a table jump to the bank function as well.

(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 RX78K0 (real-time OS) can be described. Vectors can be set (settings of interrupt request name, function name for handlers, and stack switching) with **#pragma** instruction.

(27) Interrupt handler qualifier for RTOS

This qualifier allows the interrupt handler description and the vector setting for the RX78K0 (real-time OS) made in separate files.

(28) Task function for RTOS

Specified functions are interpreted as the tasks for the RX78K0 (real-time OS) by **#pragma** instruction. This allows the description of task function for RTOS with better code-efficiency in the C source level.

(29) Static model

Specifying the **-SM** option during compilation enables the shortening of object codes, improvement of execution speed, realization of high-speed interrupt processing, and saving of memory space.

(30) Type modification

By specifying the **-ZI** option and **-ZL** option, **int/short** types are regarded as **char** type, and **long** type is regarded as **int** type.

(31) Changing function call interface

Arguments can be handed over by the previous function interface specification (using the stack only with CC78K0 Ver.2.11 compatibles) by specifying the **-ZO** option during compilation.

(32) Pascal function (**_ _pascal**)

The stack correction used for placing arguments during the function call is performed on the function callee, not on the function caller. This shortens the object code when a lot of function call appears.

(33) Automatic pascal functionization of function call interface

By specifying the **-ZR** option during compilation, the **_ _pascal** attribute is added to functions other than the **norec/_ _interrupt/_ _interrupt_brk/_ _rtos_interrupt/_ _flash/_ _flashf/_ _banked 1 to 15/**variable length argument functions.

(34) Flash area allocation method

A program can be allocated to the flash area by specifying the **-ZF** option during compilation, or a program can be used in combination with the object created in the boot area without specifying the **-ZF** option.

(35) Flash area branch table

The start-up routine, allocation of interrupt function to the flash area, and function call from the boot area to the flash area are performed by specifying the first address of the flash area branch table by the **#pragma** directive.

(36) Function of function call from boot area to flash area

A function can be called from the boot area to the flash area by specifying the function name and the ID value in the flash area called from the boot area by the **#pragma** directive.

(37) Firmware ROM function

During the prototype declaration of the interface library, manipulations regarding the firmware ROM function can be described in C source level by adding the **_ _flash** attribute to the first address.

(38) Method of **int** expansion limitation of argument/return value

By specifying the **-ZB** option during compilation, the object code can be shortened and execution speed can be improved.

(39) Array offset calculation simplification method

By specifying the **-QW2** and **-QW3** options during compilation, the offset calculation code is simplified, the object code is shortened, and the execution speed is improved.

(40) Register direct reference function

Register access can be made easily by the C specification by coding this function in the source in the same format as the function call or by declaring the use of this register direct reference function by the **#pragma realregister** directive in the module.

(41) [HL+B] based indexed addressing utilization method

By specifying the **-QE** option during compilation, the object code can be shortened and execution speed improved.

(42) On-chip firmware self-programming subroutine direct call function

A register call is made easily by the C specification, by specifying this function in the same format as the function call in the source or by declaring the use of this on-chip firmware self-programming subroutine direct call function by the **#pragma hromcall** directive of the module.

(43) **_ _flashf** function

By adding the **_ _flashf** attribute to the beginning during the function declaration, when the on-chip firmware self-programming direct subroutine call function is described in this **_ _flashf** function, the code that switches to bank save/restore and register bank 3 at each call is not generated.

(44) Memory manipulation function

By #pragma inline directive, an object file is generated by the output of the standard library functions **memcpy** and **memset** with direct inline expansion instead of function call. This function can improve the execution speed.

(45) Absolute address allocation specification

By declaring **_ _directmap** in the module in which the variable to be allocated to an absolute address is to be defined, one or more variables can be allocated to the same arbitrary address.

(46) Static model expansion specification

By specifying the **-ZM** option during compilation, restrictions on existing static models can be relaxed, improving descriptiveness.

(47) Temporary variables

By specifying the **-SM** and **-ZM** options during compilation and declaring **_ _temp** for arguments and automatic variables, an area for arguments and automatic variables can be reserved.

In addition, if the sections containing arguments and those containing automatic variables are clearly identified and the **_ _temp** declaration is applied to variables that do not require a guaranteed value match before and after a function call, memory can be reserved.

(48) Library supporting prologue/epilogue

By specifying the **-ZD** option during compilation, the prologue/epilogue code can be replaced by a library, shortening the object code.

[MEMO]

CHAPTER 2 CONSTRUCTS OF C LANGUAGE

This chapter explains the constituting elements of a C source module file.
A C source module file consists of the following tokens (distinguishable units in a sequence of characters).

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

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

`#include "expand.h"`	
`extern void testb(void);`	extern........ Keyword
`extern void chgb(void);`	
`extern bit data1;`	
`extern bit data2;`	data1, data2 .... Identifiers
`void main()`	void........ Keyword
`{`	
`    data1=1;`	1 .... Constant
`    data2=0;`	0 .... Constant
`    while(data1){`	while .... Keyword
`        data1=data2;`	{ } .... Delimiter
`        testb();`	= .... Operator
`    }`	
`    if(data1&&data2){`	if........ Keyword
`        chgb();`	&&........ Operator
`    }`	( )........ Operator
`}`	
`void lprintf(char *s,int i)`	lprintf........ Identifier
`{`	char, int........ Keywords
`    int j;`	s, i........ Identifiers
`    char *ss;`	* .... Operator
`    j=i;`	
`    ss=s;`	
`}`	
`    .`	
`    .`	
`    .`	

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. (see ESCAPE sequences below.)

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

(2) ESCAPE sequences

Nongraphic characters used for control characters as for alert, formfeed, and such are represented by ESCAPE sequences. Each ESCAPE sequence consists of the \ sign and an alphabetic character.

Nongraphic characters represented by ESCAPE sequences are shown below.

Table 2-1 List of ESCAPE Sequences

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

(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

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

2.2 Keywords

(1) ANSI-C keywords

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

```
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 CC78K0

In this C compiler the following tokens have been added as keywords to implement its expanded functions. These tokens cannot be used as labels or variable names nor can ANSI (when an uppercase character is included, the token is not regarded as a keyword).

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

`_ _callt/callt`	Declaration of **callt** function
`_ _callf/callf`	Declaration of **callf** function
`_ _sreg/sreg`	Declaration of **sreg** variable
`noauto`	Declaration of **noauto** function
`_ _leaf/norec`	Declaration of **norec** function
`bit`	Declaration of **bit** type variable
`_ _boolean/boolean`	Declaration of **boolean** type variable
`_ _interrupt`	Hardware interrupt function
`_ _interrupt_brk`	Software interrupt function
`_ _banked 1 to 15`	Bank function
`_ _asm`	**asm** statement
`_ _rtos_interrupt`	Interrupt handler for RTOS
`_ _pascal`	Pascal function
`_ _flash`	Firmware ROM function
`_ _flashf`	**_ _flashf** function
`_ _directmap`	Absolute address allocation specification
`_ _temp`	Temporary variable
`_ _mxcall`	**_ _mxcall** function[Note]

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

2.3 Identifiers

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

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

Each identifier can consist of uppercase letters, lowercase letters, or numeric characters including underscores. The following characters can be used as identifiers:

There is no restriction for the maximum length of the identifier. In this compiler, however, only the first 249 characters can be identified (refer to **Table 1-1**).

```
_(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 prototype scope

void main(void)
{
      int cot;                   ——— cot ........... Block scope
        data1=1;
        data2=0;

        while(data1){
                data1=data2;
             j1:                 ——— j1 ........... Function scope
                testb(cot);
        }
}
void testb(int x)                ——— x ........... Block scope
{
     .
     .
     .
```

(1) Function scope

Function scope refers to the entirety within a function. An identifier with function scope can be referenced from anywhere within a specified function.

Identifiers that have function scope are label names only.

(2) File scope

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

(3) Block scope

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

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

(4) Function prototype scope

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

2.3.2 Linkage of identifiers

The linkage of an identifier refers to that the same identifier declared more than once in different scopes or in the same scope can be referenced as the same object or function. An identifier by being linked is regarded to be one and the same. An identifier may be linked in the following 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 have external linkage examples:

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

(2) Internal linkage

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

The following identifier has an internal linkage example:

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

(3) No linkage

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

Examples of identifiers that have no linkage are as follows:

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

2.3.3 Name space for identifiers

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

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

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

(2) Automatic storage duration

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

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

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

Objects that have automatic storage duration are as follows:

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

2.3.5 Data types

A **type** determines the meaning of a 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 which indicates an object with size information
- Function type .... Type which indicates a function
- Incomplete type.. Type which indicates an object without size information

- Basic types (Arithmetic types)
  - Integral types
    - **char** type
    - Signed integral types
      - **signed char**
      - **short int**
      - **int**
      - **long int**
    - Unsigned integral types (specified by **unsigned**)
    - Enumeration type
  - Floating-point types
    - **float**
    - **double**
    - **long double**
- Character types
  - **char**
  - **signed char**
  - **unsigned char**
- Incomplete types
  - Array with an indefinite object size, structure, union, and **void** type
- Derived types
  - Array type (Aggregate type)
  - Structure type (Aggregate type)
  - Union type
  - Function type
  - Pointer type
- Scalar types
  - Basic (Arithmetic types)
  - Pointer type

(1) Basic types

A collection of basic data types is also referred to as “arithmetic types”. The arithmetic types consist of integral types and floating-point type.

(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** type has a sufficient size to store any character in the basic execution character set. The value of a character to be stored in a **char** type object becomes positive. Data other than characters is handled as an unsigned integer. In this case, however, if an overflow occurs, the overflowed part will be ignored.

(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 qualifier.

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

A signed integral type data has its corresponding unsigned integral type data. Both share an area of the same size. The positive number of a signed integral type data is a partial collection of unsigned integral type data.

(iii) Unsigned integral type

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

(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 the three types.

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

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

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

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

- Format

The floating-point number format is shown below.

(Higher address)	s	e	m	(Lower address)
	31	30 23	22 0	

The numerical values in this format are as follows.

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

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.

$$(-1)^{\text{(Value of sign)}} * 0$$

- 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 the element type. Member objects all have an area of the same size. The array type specifies the number of element types and the elements of the array. It cannot create the array of incomplete type.

(ii) Structure type

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

(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. A function type data specifies the type of return value, the number of parameters, and the type of parameter. If the type of return value is T, the function is referred to as a function that returns T.

(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 arithmetic types (basic type) and pointer type are collectively called the scalar types. The scalar types include the following data types:

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

2.3.6 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 which has the parameter type list.
- If both types have a parameter type list (i.e., functions with prototypes), the composite type is one with a prototype consisting of all information that can be combined from the 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, which does not change in value during the execution of the program, and its value must be set beforehand. A type for each constant is determined according to the format and value specified for the constant. The following 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 the decimal point nor exponent part. An unsigned suffix can be added after the integer constant to indicate that the integer constant is unsigned. A long suffix can be added after the integer constant to indicate that the integer constant is long.

There are the following three types of integer constant.

- 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 the base (radix) of 10 and must begin with a number other than 0 followed by any numbers 0 through 9 (example: 56U).

(2) Octal constant

An octal constant is an integer value with the 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 one or more character strings enclosed in a pair of single quotes as in ‘X’ or ‘ab’.

A character constant does not include single quote’, back slash (¥ 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 Literal

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

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

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

2.6 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 preprocessor directives. (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.

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

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

2.8 Header Name

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

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

```
#include <header name>
#include “header name”
```

2.9 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 by the **-ZP** option.

Example
```
/* comment statement */
//comment statement
```

[MEMO]

This chapter explains how data (variables) or functions to be used in C should be declared as well as scope for each data or function. A declaration means the specification of an interpretation or attribute for an identifier or a collection of identifiers. A declaration to reserve a storage area for an object or function named by an identifier is referred to as a “definition”.

An example of a declaration is shown below.

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

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

3.1 Storage Class Specifiers

A storage class specifier specifies the storage class of an object. It indicates the storage location of a value, which the object has, and the scope of the object. In a declaration, only 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 typedef** for details of the **typedef** specifier.

(2) extern

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

(3) static

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

(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, which 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`**

The followings explain 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)Note
- `double` .... Double precision floating point number (1.17549435E–38F to 3.40282347E+38F)Note
- `long double` .... Extended precision floating point number (1.17549435E–38F to 3.40282347E+38F)Note
- structure/union specifier .... Collection of member objects
- enumeration specifier .... Collection of **int** type constants
- `typedef` name .... Synonym of specified type
- `bit, boolean, _ _boolean` .... Integers represented with a single bit (0 to 1)

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

Note Range of absolute values

3.2.1 Structure specifier and union specifier

Both the structure specifier and union specifier indicate a collection of named members (objects). These member objects can have different types from one another.

(1) Structure specifier

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

However, because the 78K0 Series contains a restriction whereby word data is unable to be read from or written to odd addresses, the code size is prioritized by default, and align data is inserted to ensure members of 2 bytes or more are allocated to even addresses. Gaps may therefore occur between members due to the align data.

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

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

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

Example of structure declaration

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

(2) Union specifier

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

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

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

Example of union declaration

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

Each member object can be any type other than the incomplete types or function types. The member can declare with the number of bits specified. The member with the number of bits specified is called a bit field.

In this compiler, extended functions related to bit field declaration have been added. For the details, refer to **11.5 (15) 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.

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

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

The following shows an example of 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 declares as shown below.

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

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

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

```
enum hue{
       chartreuse,
       burgundy,
       claret=20,
       winedark
};
enum hue col,*cp;
void main(void) {
      col=claret;
      cp=&col ;
      /*...*/ (*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.

An 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.

```
structure/union    identifier
```

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

In the following example, the tag "tnode" specifies a structure that includes pointers to an integer and 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 Lvalues only.

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

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

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

No type qualifier can be included in the specification of a function type. However, **callt**, **_ _ callt**, **callf**, **_ _ callf**, **noauto**, **norec**, **_ _ leaf**, **_ _ interrupt**, **_ _ interrupt_brk**, **_ _rtos_interrupt**, **_ _pascal**, which are the type qualifiers unique to this compiler mentioned in **2.2 Keywords**, can be included as type qualifiers.

sreg, **_ _sreg**, **_ _directmap**, and **_ _temp** are also type qualifiers.

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

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

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

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

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

3.4 Declarators

A declarator declares an identifier. Here, pointer declarators, array declarators, and function declarators are mainly discussed. By a declarator, the scope of an identifier and a function or object which has a storage duration and a type are determined.

The description of each declarator is shown below.

3.4.1 Pointer declarators

A pointer declarator indicates that an identifier to be declared is a pointer. A pointer points to (indicates) the location where a value is stored. Pointer declarations are performed as follows.

```
* type qualifier list    identifier
```

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

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

3.5 Type Names

A type name is the name of a data type that indicates the size of a function or object. Syntax-wise, it is a function or object declaration less identifiers.

Examples of type names are given below.

- int .... Specifies an **int** type.
- int * .... Specifies a pointer to an **int** type.
- int *[3] .... Specifies an array which has three pointers to an **int** type.
- int (*) [3] .... Specifies a pointer to an array which has three **int** types.
- int *( ) .... Specifies a function which returns a pointer to an **int** type which has no parameter specification.
- int (*) (void) .... Specifies a pointer to a function which returns an **int** type which no parameter specification.
- int (*const [ ]) (unsigned int, ...) ... Specifies an indefinite number of arrays which have 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, **typedef** name **t** is declared with signed int type, and **typedef** name **plain** is declared with **int** type, respectively, and the 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 the point that the first bit field declaration has unsigned as the type specifier (therefore, **t** becomes the name of the structure member), and the second bit field declaration, on the other hand, has **const** as the type qualifier (qualifiers **t** which can be referred to as **typedef** name). After this declaration, if:

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

is found within the effective range, the function **f** is declared as “function which has one parameter and returns **signed int**”, and the parameter is declared as “pointer type for the function which 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 which has a static storage duration will be initialized to a null pointer constant.

Example

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

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

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

Example

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

(3) Initialization of character arrays

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

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

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

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

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

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

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

(4) Initialization of aggregate or union type objects

- Aggregate type
  An aggregate type object is initialized with a list of initializers described in ascending order of subscripts or members. The initializer list to be specified must be enclosed in braces.
  If the number of initializers in the list is less than the number of aggregate members, the members not covered by the initializers will be implicitly initialized just the same as an object which has a static storage duration.
  With an array with an unknown size, the number of its elements is governed by the number of initializers and the array will no longer become an incomplete type.
- Union type
  A union type object is initialized with an initializer for the first member of the union that is enclosed in braces.

In the following example, the array “x” with an unknown size will change to a 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},
         }
};
```

[MEMO]

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

2. Conventions

○: 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.

General integral promotion will retain the value of the original data type including its sign.

char type data without type qualifier will normally be handled as **signed char** in this compiler. It can be handled as an **unsigned char** with option.

(2) Signed integers and unsigned integers

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

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

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

Conversion operations from signed integral type to unsigned integral type are as listed in **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.

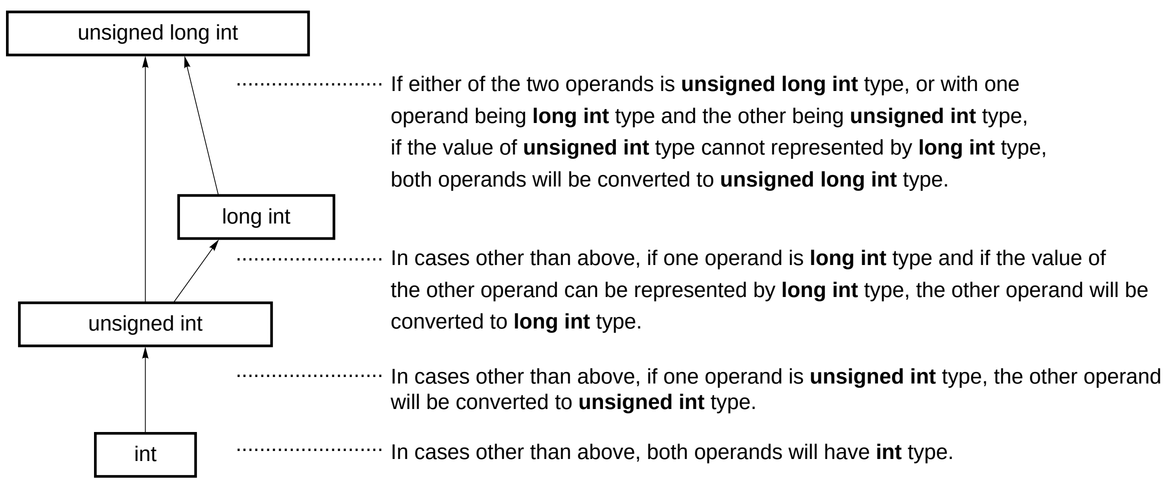
(3) Usual arithmetic type conversions

Types obtained as a result of operations on arithmetic type data will have a wide range of values.
The type conversion of the operation result is performed as follows.

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

In cases other than above, general integer expansion is performed for both operands according to the following rules. **Figure 4-1** shows the rules.

Figure 4-1 Usual Arithmetic Type Conversions



In this compiler, the conversion to **int** type can be intentionally disabled by compile condition (optimizing option) (For the details, refer to **CC78K0 C Compiler Operation User's Manual**).

4.2 Other Operands

(1) Lvalues and function locators

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

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

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

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

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

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

(2) void

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

(3) Pointers

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

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

[MEMO]

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 precedence of operators used in C.

Table 5-1 Evaluation Precedence of Operators

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

Operations in the same line contain the same priority.

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

5.1 Primary Expressions

Primary expressions include the following:

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

An identifier which becomes a primary expression is an Lvalue if an object is declared or a function locator if a function is declared. The data type of a constant is determined according to the value specified for the constant as explained in **2.4 Constants**. String literal(s) become an Lvalue that has a data type as explained in **2.5 String Literal**.

5.2 Postfix Operators

A postfix operator is an operator that appears or is placed after an object or function.
The followings explain each primary expression.

(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, subscript operators as many as the number of dimensions must be connected.

In the following example, x becomes an **int** type array of 3*5. In other words, x is an array which has 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 ≥ 2) consisting of i*j*...*k, the array can be specified with the n number of subscript operators. In this case, E becomes a pointer to an array of (n – 1)th dimension consisting of j*...*k.

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 passed to the function are indicated in parentheses ( ).

The description related to function includes the function prototype declaration, the function definition (the body of a function), and the function call. The function prototype declaration specifies the value a function returns, the type of argument, and the storage class.

If the function prototype declaration is not referred to in a function call, each argument is extended with general integer. This is called “default actual argument extension”. Performing a function prototype declaration avoids default actual argument extension and detects the mistakes of the type and number of argument and the type of return value.

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

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

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 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 by taking the data type of the object into account.

SYNTAX

```
postfix-expression ++
```

<2> Postfix – – (Decrement) operator

FUNCTION

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

SYNTAX

```
postfix expression --
```

NOTE

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

5.3 Unary Operators

A unary operator performs an operation on 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.

(1) Prefix increment/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 an Lvalue referring to an object not declared with the register storage class specifier. Neither a function locator nor a bit field can be used as the operand of this unary operator.

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

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

Unary Expressions + – ~ !

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 an Lvalue which 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 which 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 multiplying operators and adding operators. Multiplying operators find the product, quotient, and remainder of two operands. Adding operators find the sum and difference of two operands.

- Multiplying operators `*` `/` `%`
- Adding operators `+` `-`

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

a/b		b: +	b: –
a	+	+	–
	–	–	+

a % b		b: +	b: –
a	+	+	+
	–	–	–

Remark a, b indicates each operand.

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. The following explain multiplying operators and adding operators. 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, b indicates each operand.

The followings explain each shift operator. 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 have 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 followings explain relational operators and equality operators. E1 and E2 used in the explanation of syntax indicate operand and expression.

(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
```

Relational Operator < > <= >=

<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 both 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.

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

The followings explain bitwise logical operators. 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 which returns an integral value that has “1” bits in positions where both operands have “1” bits and that has “0” bits everywhere else.

The bitwise AND operator must be specified with an “operator”.

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 which returns an integral value that has a “1” bit in each position where exactly one of the operands has a “1” bit and that has a “0” bit in each position where both operands have a “1” bit or both have a “0” bit.

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 which returns an integral value that has a “1” bit in each position where at least one of the operands has a “1” bit and that has a “0” bit in each position where both operands have a “0” bit.

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 an operand an expression.

(1) Logical AND operators

Logical AND Operators &&

FUNCTION

The && operator performs 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 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 Each Bit in 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 followings explain conditional operators.

Conditional Operators ? :

FUNCTION

If the value of the first operand is nonzero, it evaluates the second operand before the colon. If the value of the first operand is zero, it evaluates the third operand after the colon. The result of the entire conditional expression will be the value of the second or third operand.

SYNTAX

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

EXAMPLE

```
#define TRUE  1
#define FALSE 0
char flag;
int ret;
int 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 **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 followings explain each assignment operator. 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 (Lvalue) before the value is stored.

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

```
int f(void);

char c;
/*...*/ ((c=f())==-1) /*...*/
```

SYNTAX

```
E1 = E2
```

(2) Compound assignment operators

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 Lvalue (left operand). The compound assignment expression “E1 op = E2” (where op indicates a suitable binary operator) is equivalent to the simple assignment expression “E1 = E1 op (E2)”, except that the Lvalue (E1) is only evaluated once. The following compound assignment expressions will produce the same result as the respective simple assignment expressions on the right.

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

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

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 time instead of execution time.

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

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

(1) General integral constant expression

A general integral constant expression has a general integral type. The following operands may be used:

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

(2) Arithmetic constant expression

An arithmetic constant expression has an integral type. The following operands may be used:

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

(3) Address constant expression

An address constant expression is a pointer to an object that has a static storage duration or a pointer to a function locator. Such an expression must be created explicitly using the unary & operator or implicitly using an expression with an array type or function type. The following operands may be used:

- 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.

[MEMO]

CHAPTER 6 CONTROL STRUCTURES OF C LANGUAGE

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

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

(1) Sequential processing

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

(2) Conditional control (selection) processing

According to the status of the program under execution, the next executable statement is selected and executed. The selection condition is specified in a control statement. The control statement determines which of the 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 during the condition indicated by the control statement.

(4) Branch processing

C is caused to exit from the current program flow and control is transferred to a specified label. Program execution starts from the statement next to the specified label.

There are six types of statements used in C.

• Labeled statement	Causes branch according to the value of switch statement and the destination of 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 the control expression becomes equal to 0.
• Branch statement................................	Causes unconditional branch to different destination

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

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

6.1 Labeled Statements

A labeled statement specifies the destination of **switch** or **goto** statement. The **switch** statement selects the statement specified by a control expression from among statements with two or more options. The labeled statement becomes the label of the statement to be executed by the **switch** statement. The **goto** statement causes unconditional branching to the applicable label from the normal flow of processing.

The 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 to break out of the **switch** body. If the return value of f( ) is other than 1 to 3, the expression that follows the **default** label is executed. In this case, the value of i becomes 1.

6.2 Compound Statements or Blocks

A compound statement or block is synonymous to each other and consists of two or more statements grouped together with enclosing braces and executed as 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 do not need any body.

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

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

```
int p(int) ;
void 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 (}) which closes a compound statement as shown below.

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

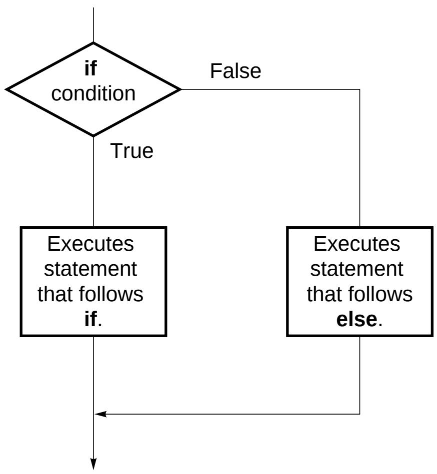
6.4 Conditional Control Statements

Conditional control (or selection) statements include **if** and **switch** statements. The **if** or **switch** statement allows the program to choose one of several groups of statements to execute, based on the value of the control expression enclosed in parentheses.

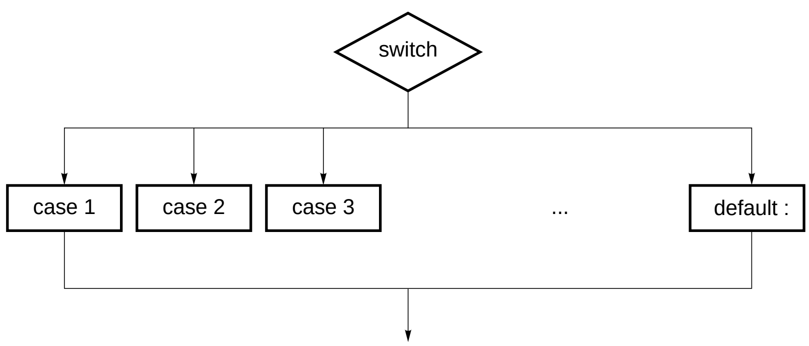
The control flows of **if** and **switch** statements are illustrated in **Figure 6-1** below.

Figure 6-1 Control Flows of Conditional Control Statements

Control flow of **if** statement



Control flow of **switch** statement



(1) if and if ... else statements

Conditional Control Statements — if, if ... else

FUNCTION

An **if** statement executes the statement that follows the control expression enclosed in parentheses if the value of the control expression is nonzero.

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

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 **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 Control 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.

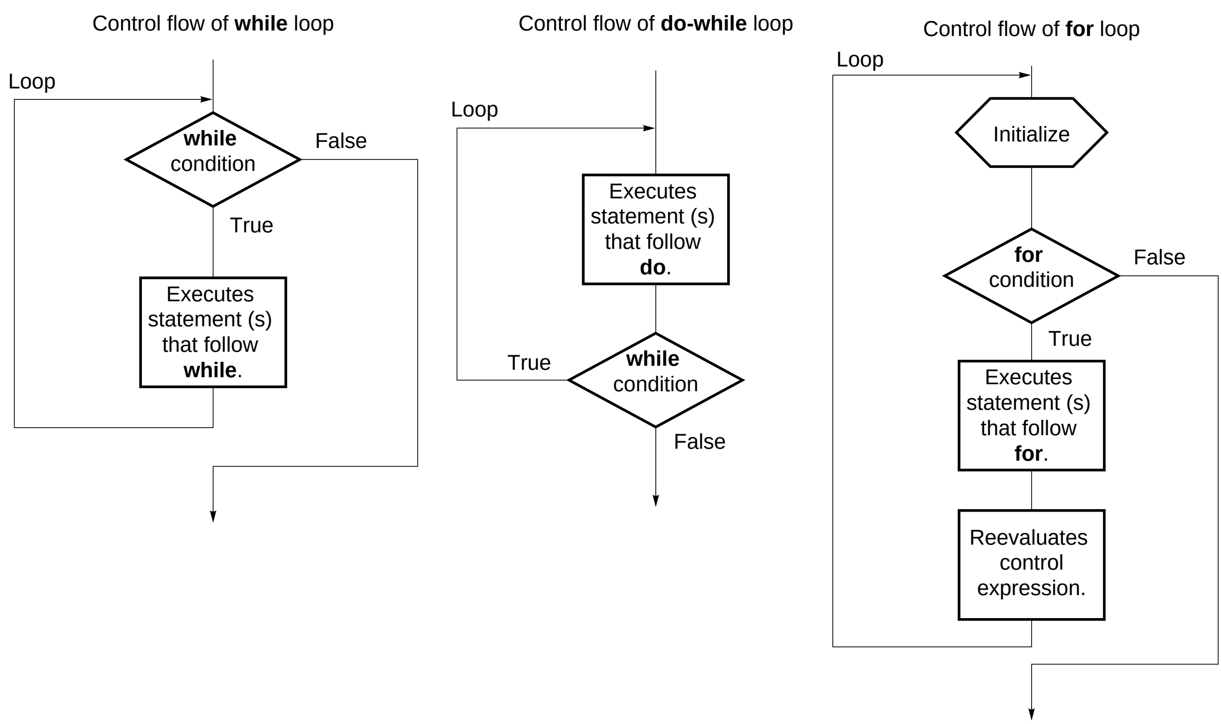
6.5 Looping Statements

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

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

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

Figure 6-2 Control Flows of Looping Statements



(1) while statement

Looping 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 brace brackets are the body of this **while** loop. The control expression "i<11" returns 0 if the value of i becomes 11. For this reason, the loop body is executed repeatedly as long as the value of i is less than 11 (between 1 and 10). "**while**(1) {statement}" is used to endlessly perform a loop statement.

(2) do statement

Looping Statements do statement

FUNCTION

A **do** statement executes the body of the loop and then tests the control expression enclosed in parentheses to see if its value is True (nonzero). The **do** statement evaluates the control expression after the loop body has been executed.

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 brace brackets are the body of this **do ... while** loop. The control expression “i<11” returns 0 if the value of i becomes 11. For this reason, the loop body is executed repeatedly as long as the value of i is less than 11 (between 1 and 10). The body of the loop is always performed once or more since the control expression of a **do** statement is evaluated after execution.

(3) for statement

Looping 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 execute only once in the beginning of the loop, the second is the control expression for testing the counter value, and the third is a step statement executed in the end of every loop and reevaluate the variable after the execution.

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.

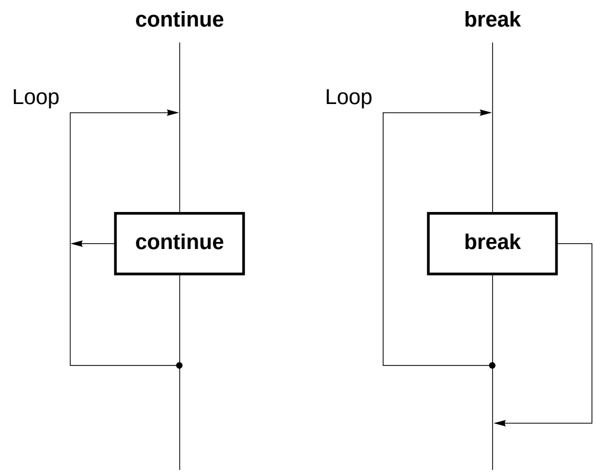
6.6 Branch Statements

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 without condition and transfers control to the statement next to “point”.

NOTE

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

(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 a 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 it calls and returns the value of the **return** statement expression as the value of the function call expression. Two or more **return** statements may be used in a function. Using the closing brace bracket " } " at the end of a function produces the same result as when a **return** statement without expression is executed.

SYNTAX

```
return expression ;
```

EXAMPLE

```
int f(int);

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

int f(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.

[MEMO]

CHAPTER 7 STRUCTURES AND UNIONS

A structure or union is a collection of member objects that have different types and 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 name called a tag name can be given to the structure declaration list.

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

[Declaration of structure]

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

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

[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 which specifies each member in bits can also be specified.

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

[Example of structure declaration list]

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

[Example of bit field declaration]

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

(3) Arrays and pointers

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

[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 which share the same memory space (or overlap in memory).

(1) Declaration of union and union variable

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

[Declaration of union]

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

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

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

(2) Union declaration list

A union declaration list specifies the structure of a union type to be declared. Each element on the union declaration list is called a member. Declared members are allocated to the same area. In the following [Example of union declaration list], an area is allocated to ‘c’, which becomes the largest area of the members. The other members are not allocated new areas but use the same area.

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

Each member can have any object type other than the above 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 declaration come after the preprocessing. These declaration are referred to as "external declaration" because they appear outside a function and have effective file ranges.

A declaration to give a name to external objects by identifiers or a declaration to secure storage for a function is called an external definition. If an identifier declared with external linkage is used in an expression (except the operand part of the **sizeof** operator), only 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 forcedly converted by the compiler. By using the form of an identifier list for the parameters, this type conversion can be described. An example of this identifier list is shown below.

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

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

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

Examples of external object definitions are shown 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 PREPROCESSOR DIRECTIVES (COMPILER DIRECTIVES)

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

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

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

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

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

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

[Example]

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

```
#define SYM 0

#if defined SYM
    .
    .
    .
#endif
```

(1) #if directive

Conditional Translation #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  new-line「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 #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 new-line「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 #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 new-line「group」
```

EXAMPLE

```
#define ON
#ifdef ON
    .
    .
    .
#endif
```

EXPLANATION

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

(4) #ifndef directive

Conditional Translation #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 new-line ⌈group⌋
```

EXAMPLE

```
#define ON
#ifndef ON
    .
    .
    .
#endif
```

EXPLANATION

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

(5) #else directive

Conditional Translation #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 new-line ｢group｣
```

EXAMPLE

```
#define ON
#ifdef ON
    .
    .
    .
#else
    .
    .
    .
#endif
```

EXPLANATION

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

(6) #endif directive

Conditional Translation #endif

FUNCTION

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

SYNTAX

```
#endif new-line
```

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

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

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

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

(1) #include < > directive

Source File Inclusion #include< >

FUNCTION

If the directive form is **#include** < >, the C compiler searches the directory specified with **-i** compiler option, directory specified by the INC78K0 environment variable, and directory \NECTools32\INC78K0 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> new-line
```

EXAMPLE

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

EXPLANATION

In the above example, the C compiler searches the directory specified by the INC78K0 environment variable and directory \NECTools32\INC78K0 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 “ ” directive

Source File Inclusion #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 with **-i** compiler option, directory specified by the INC78K0 environment variable, and directory \NECTools32\INC78K0 registered in the registry is searched. Then, the compiler replaces the **#include** directive line with the entire contents of the specified file thus searched.

SYNTAX

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

EXAMPLE

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

EXPLANATION

In the above example, the C compiler searches the current working directory, the directory specified by the INC78K0 environment variable, and directory \NECTools32\INC78K0 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 directive

Source File Inclusion #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  new-line
```

EXAMPLE

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

EXPLANATION

In the inclusion of other source files with the directive form: **#include** preprocessing token string, 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 with -i compiler option, directory specified by the INC78K0 environment variable, and directory \NECTools32\INC78K0 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 with **-i** compiler option, directory specified by the INC78K0 environment variable, and directory \NECTools32\INC78K0 registered in the registry is searched.

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 new-line

- Function format
  #define identifier （「identifier-list」） replacement-list new-line

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

(5) Scope of macro definition

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

(6) #define directive

Macro Replacement #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 new-line) whenever the same identifier appears in the source code after the definition by this directive.

SYNTAX

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

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 #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 new-line). 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 new-line
```

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 safety' sake, be sure to enclose the replacement list in parentheses, because unlike a function definition, this function-form macro is merely to replace a sequence of characters.

(8) #undef directive

Macro Replacement #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 new-line
```

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 preprocessor directive for line control **#line** replaces the line number to be used by the C compiler in translation with the number specified in this directive. If a string (character string) is given in addition to the number, the directive also replaces the source file name the C compiler has with the specified string.

(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  new-line
```

[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"  new-line
```

[Example]

```
#line 10 "file1.c"
```

(3) To change using preprocessor token string

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

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

[Example]

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

9.5 #error Preprocess Directive

#error preprocess directive is a directive that outputs a message including the specified preprocessor tokens and incompletely terminates a compile. This preprocessor is used to terminate a compile.

This preprocessor is specified as follows.

```
#error "preprocessing-token-string" new-line
```

[Example]

In this example, the macro name **_ _K0 _ _**, which indicates the device series that this compiler has, is used. If the device is the 78K0 Series, the program between **#if** and **#else** is compiled. In the other cases, the program between **#else** and **#endif** is compiled, but the compile will be terminated with an error message “not for 78K0” output by **#error** directive.

```
#if _ _K0_ _
    .
    .
    .
#else
#error "not for 78K0"
    .
    .
    .
#endif
```

9.6 #pragma Directives

#pragma directive is a directive to instruct the compiler to operate in the compiler definition method. In this compiler, several **#pragma** directives to generate codes for the 78K0 Series (For the details of **#pragma** directives, refer to **CHAPTER 11 EXTENDED FUNCTIONS)**.

[Example]

In this example, **#pragma NOP** directive enables the description to directly output a **NOP** instruction in the C source.

```
#pragma NOP
```

9.7 Null Directives

Source lines that contain only the **#** character and white space are called null directives. Null directives are simply discarded during preprocessing. In other words, these directives have no effect on the compiler. The syntax of null directives is given below.

```
# new-line
```

9.8 Compiler-Defined Macro Names

In 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 preprocessor directive must not be applied to these macro name and defined identifiers. All the macro names of the compiler definition start with underscore followed by an uppercase character or the second underscore.

In addition to the above macro names, macro names indicating the series names of devices depending on the device subject to applied product development and macro names indicating device names are provided. To output the object code for the target device, these macro names must be specified by the option at compilation time or by the processor type in the C source.

- Macro name indicating the series names of devices
  `'_ _K0_ _'`
- Macro name indicating the device name
  ‘_ _’ is added before the device type name and ‘_’ is added after the device type name.
  Describe English characters in uppercase.
  (Example) `_ _054_` `_ _054Y_`

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

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

- Define as follows when the static model is specified

```
#define _ _STATIC_MODEL_ _  1
```

The device type for compile is specified by adding the followings to the command line
‘-c device type name’

(Example) `cc78k0 -c054Y prime.c`

The device type does not need to be specified on compile by specifying it at the start of the C source program.

'#pragma PC (device type)'

(Example)
```
#pragma PC(054Y)
        .
        .
        .
```

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

- Comment statement
- Preprocessor directives that do not generate definition/reference of variables nor functions.

CHAPTER 10 LIBRARY FUNCTIONS

C has no instructions to transfer (input or output) data to and from external sources (peripheral devices and equipment). This is because of the C language designer's intent to hold the functions of C to a minimum. However, for actually developing a system, I/O operations are requisite. Thus, this C compiler 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 to 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 by a stack and a register, respectively. However, when the old function interface supporting option (**-ZO**) is not specified in the normal model, the first argument is, if possible, also passed by the register. In addition, all of the arguments are passed by the register in the static model.

For the **-ZO** option, refer to the **CC78K0 C Compiler Operation User's Manual**.

10.1.1 Arguments

Placing or removing arguments on or from the stack is performed by the caller (calling side). The callee (called side) only references the argument values. However, when the argument is passed by the register, the callee directly refers to the register and copies the value of the argument to another register, if necessary. 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 on the stack.

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

The minimum unit of data can be stacked is 16 bits. A data type larger than 16 bits is stacked in units of 16 bits one by one from its MSB. An 8-bit type data is extended to a 16-bit type data for stacking.

When in static model, all of arguments are passed by the register.

Maximum of 3 arguments and a total of 6 bytes can be passed. Passing the float, double, and structure arguments is not supported.

The following shows the list of the passing of the first argument. The second argument and thereafter is passed via a stack in the normal model.

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 (Normal Model)

Type of First Argument	Passing Method (without -ZO Specification)	Passing Method (with -ZO Specification)
1-byte, 2-byte integers	AX	Passed via a stack
3-byte integer	AX, BC	Passed via a stack
4-byte integer	AX, BC	Passed via a stack
Floating-point number (**float** type)	AX, BC	Passed via a stack
Floating-point number (**double** type)	AX, BC	Passed via a stack
Others	Passed via a stack	Passed via a stack

Remark Of the types shown above, 1- to 4-byte integers include structures and unions.

Table 10-2 List of Passing Arguments (Static Model)

Type of Argument	1st Argument	2nd Argument	3rd Argument
1-byte integer	A	B	H
2-byte integer	AX	BC	HL

Remark If the arguments are a total of 4 bytes, some of the arguments are allocated to AX and BC, and the rest to HL or H.
1- to 4-byte integers do not include structures and unions.

10.1.2 Return values

The return value of a function is stored in units of 16 bits starting from its LSB in the direction from the register BC to the register DE. When returning a structure, the first address of the structure is stored in the register BC. When returning a pointer, the first address of the structure is stored in the register BC.

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

Table 10-3 List of Storing Return Value

(1) Normal model

Type of Return Value	Method of Storing
1 bit	CY
1-byte, 2-byte integers	BC
4-byte integer	BC (low-order), DE (high-order)
Floating-point number (**float** type)	BC (low-order), DE (high-order)
Floating-point number (**double** type)	BC (low-order), DE (high-order)
Structure	Copies the structure to return to the area specific to the function and stores the address to BC
Pointer	BC

(2) Static model

Type of Return Value	Method of Storing
1 bit	CY
1-byte integer	A
2-byte integer	AX
4-byte integer	AX (low-order), BC (high-order)
Pointer	AX

10.1.3 Saving registers to be used by individual libraries

Library that uses HL (when in normal model) and DE(when in static model) 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) No -ZR option specified

The procedure of passing arguments and return value is shown below.

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

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

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

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

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

<5> Processing **func** and storing the return value in registers (by the callee)
The low-order 16 bits of the return value “long” are stored in BC and the high-order 16 bits of the return value, in DE.

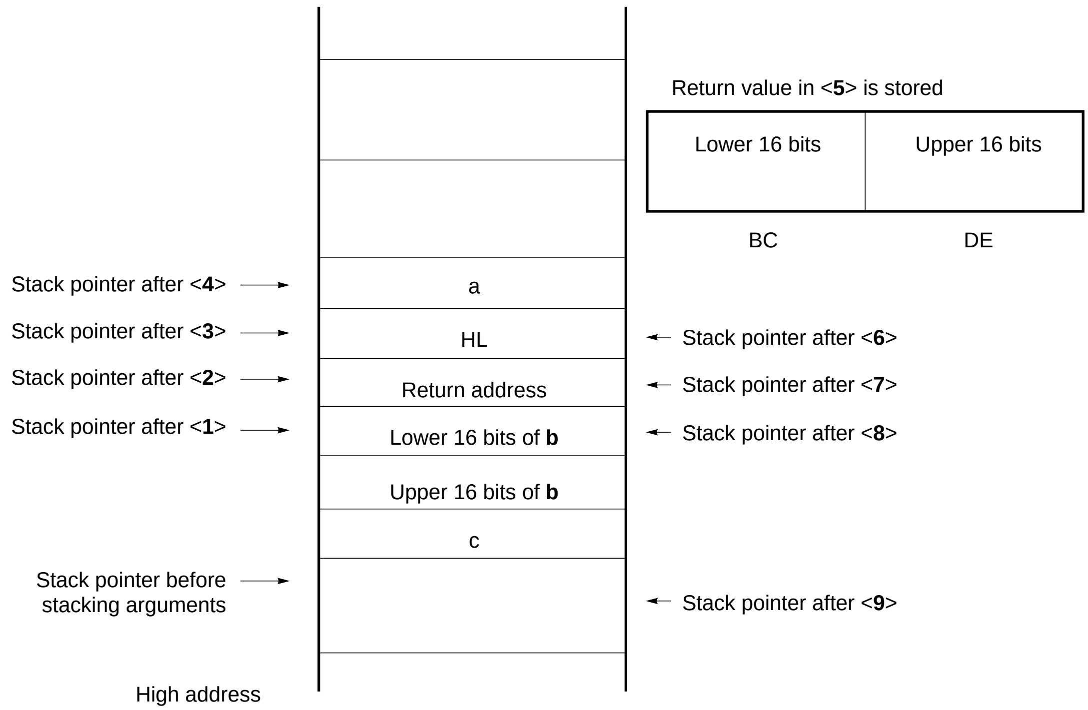
<6> Restoring the stored first argument (by the callee)

<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)



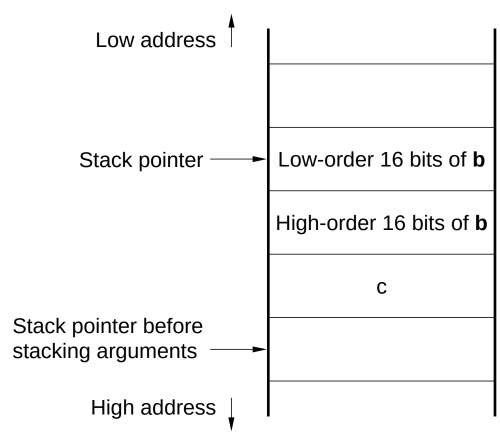
(2) If the -ZR option is specified

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);"`

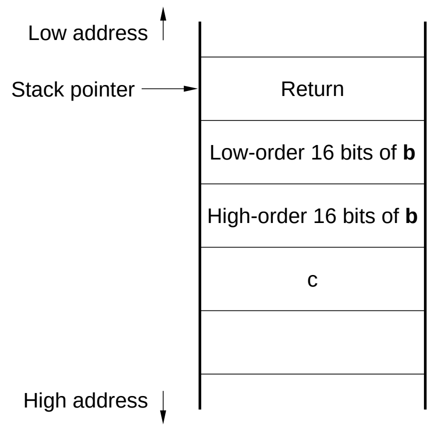
<1> Placing arguments on the stack (by the caller)

The high-order 16 bits of arguments "c" and "b" and the low-order 16 bits of argument "b" are placed on the stack in that order. a is passed by AX register.

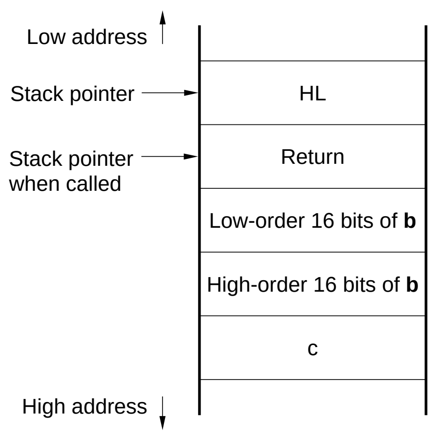


<2> Calling **func** by a **call** instruction (by the caller)

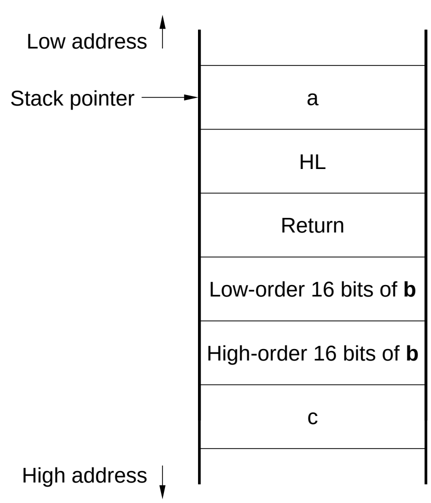
Control is transferred to the function **func** when the stack is in the state shown below.



<3> Saving the register used (by the callee)



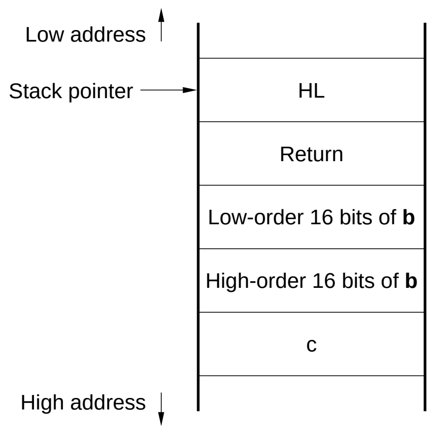
<4> The first argument called by the register is placed on the stack



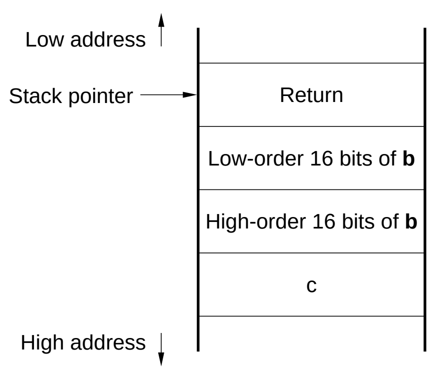
<5> Performing processing of the function **func**, and storing return values in the register (by the callee)
The low-order 16 bits of the return value are stored in BC and the high-order 16 bits are stored in DE.

BC	DE
Low-order 16 bits of return value	High-order 16 bits of return value

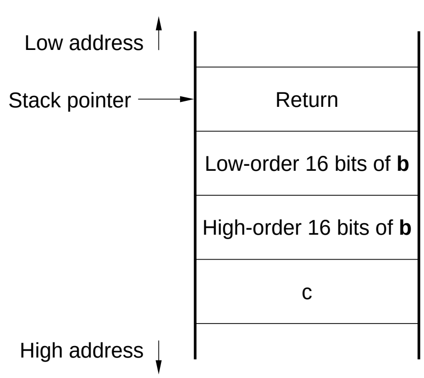
<6> Restoring the first placed argument (by the callee)



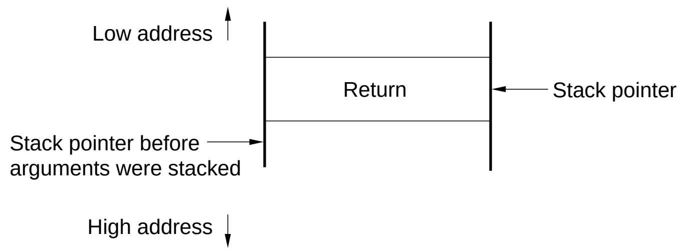
<7> Restoring the saved registers (by the callee)



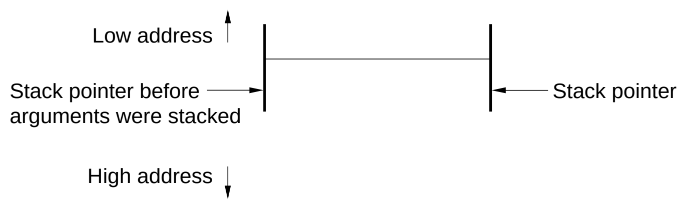
<8> Storing the return address in a register, moving the value of the stack pointer to the position where the argument is pushed to the stack, and removing the argument from the stack (on the called side).



<9> Restoring the return address stored in the register (by the callee)



<10> Returning control to the functions on the caller by the **ret** instruction (by the callee)



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.

The headers of this C compiler 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
```

Caution The functions to be supported differ depending on the memory models (normal model and static model). Also, functions that operate during normal operation differ depending on the -ZI and -ZL options. For functions that do not operate normally because of the existence of -ZI and -ZL options, a warning message “The prototype declaration is not performed” is output.

(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 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. The function to be declared differs depending on the options and the specification models.

Table 10-4 Contents of ctype.h

Existence of -ZI, or -ZL Specification / Function Name	Normal Model				Static Model			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
isalnum	√	√	√	√	√	—	√	—
isalpha	√	√	√	√	√	—	√	—
iscntrl	√	√	√	√	√	—	√	—
isdigit	√	√	√	√	√	—	√	—
isgraph	√	√	√	√	√	—	√	—
islower	√	√	√	√	√	—	√	—
isprint	√	√	√	√	√	—	√	—
ispunct	√	√	√	√	√	—	√	—
isspace	√	√	√	√	√	—	√	—
isupper	√	√	√	√	√	—	√	—
isxdigit	√	√	√	√	√	—	√	—
tolower	√	√	√	√	√	—	√	—
toupper	√	√	√	√	√	—	√	—
isascii	√	√	√	√	√	—	√	—
toascii	√	√	√	√	√	—	√	—
_toupper	√	√	√	√	√	—	√	—
_tolower	√	√	√	√	√	—	√	—
tolow	√	√	√	√	√	—	√	—
toup	√	√	√	√	√	—	√	—

√: Supported

—: Not supported

(2) setjmp.h

This header is used to define program control functions. In this header, the following functions are defined. The function to be declared differs depending on the option and the specification models.

Table 10-5 Contents of setjmp.h

Function Name \ Existence of -ZI, or -ZL Specification	Normal Model				Static Model			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
setjmp	√	√	√	√	√	—	√	—
longjmp	√	√	√	√	√	—	√	—

√: Supported

—: Not supported

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

[Declaration of **int** array type **jmp_buf**]

• Normal model

```
typedef int jmp_buf[11]
```

• Static model

```
typedef int jmp_buf[3]
```

(3) stdarg.h (normal model only)

This header used to define special functions. In this header, the following three functions have been defined:

Table 10-6 Contents of stdarg.h

Function Name \ Existence of -ZI, or -ZL Specification	Normal Model			
	None	ZI	ZL	ZI ZL
va_arg	√	√	√	√
va_start	Δ	Δ	Δ	Δ
va_starttop	Δ	Δ	Δ	Δ
va_end	√	√	√	√

√: Supported

Δ: Operation is guaranteed, however there are limitations

In the header **stdarg.h** the following object has been declared:

[Declaration of pointer type **va_list** to **char**]

```
typedef char *va_list;
```

(4) stdio.h

This header is used to define I/O functions. In this header, next functions have been defined.
The function to be declared differs depending on the options and the specification models.

Table 10-7 Contents of stdio.h

Function Name \ Existence of -ZI, or -ZL Specification	Normal Model				Static Model			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
sprintf	√	×	√	×	—	—	—	—
sscanf	√	×	√	×	—	—	—	—
printf	√	×	√	×	—	—	—	—
scanf	√	×	√	×	—	—	—	—
vprintf	√	×	√	×	—	—	—	—
vsprintf	√	×	√	×	—	—	—	—
getchar	√	√	√	√	√	—	√	—
gets	√	√	√	√	√	√	√	√
putchar	√	√	√	√	√	—	√	—
puts	√	√	√	√	√	—	√	—

√: Supported

×: Operation is not guaranteed

—: Not supported

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 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, these functions are not defined.

Table 10-8 Contents of stdlib.h

Existence of -ZI, or -ZL Specification / Function Name	Normal Model				Static Model			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
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	√	√	—	—	—	—	—	—

√: Supported
×: Operation is not guaranteed
—: Not supported

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** (except static model)]

```
typedef struct{
    int quot;
    int rem;
}div_t;
```

[Declaration of structure type **ldiv_t** which has **long int** type members **quot** and **rem** (except when -ZL is specified in static model and normal model)]

```
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 header, the following functions have been defined. Function to be defined differs depending on the options and specification models.

Table 10-9 Contents of string.h

Existence of -ZI, or -ZL Specification / Function Name	Normal Model				Static Model			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
memcpy	√	√	√	√	√	—	√	—
memmove	√	√	√	√	√	—	√	—
strcpy	√	√	√	√	√	√	√	√
strncpy	√	√	√	√	√	—	√	—
strcat	√	√	√	√	√	√	√	√
strncat	√	√	√	√	√	—	√	—
memcmp	√	×	√	×	√	—	√	—
strcmp	√	×	√	×	√	—	√	—
strncmp	√	×	√	×	√	—	√	—
memchr	√	√	√	√	√	—	√	—
strchr	√	√	√	√	√	—	√	—
strcspn	√	×	√	×	√	—	√	—
strpbrk	√	√	√	√	√	√	√	√
strrchr	√	√	√	√	√	—	√	—
strspn	√	×	√	×	√	—	√	—
strstr	√	√	√	√	√	√	√	√
strtok	√	√	√	√	√	√	√	√
memset	√	√	√	√	√	—	√	—
strerror	√	√	√	√	√	—	√	—
strlen	√	×	√	×	√	—	√	—
strcoll	√	×	√	×	√	—	√	—
strxfrm	√	×	√	×	√	—	√	—

√: Supported
×: Operation is not guaranteed
—: Not supported

(7) error.h

error.h includes **errno.h**.

(8) errno.h

In this header, the following objects have been defined:

[Definitions of macro names “EDOM”, “ERANGE”, and “ENOMEM”]

```
#define EDOM   1
#define ERANGE 2
#define ENOMEM 3
```

[Declaration of **volatile int** type external variable **errno**]

```
extern volatile int errno;
```

(9) limits.h

In this header, the following macro names have been defined:

```
#define CHAR_BIT          8
#define CHAR_MAX          +127
#define CHAR_MIN          -128
#define INT_MAX           +32767
#define INT_MIN           -32768
#define LONG_MAX          +2147483647
#define LONG_MIN          -2147483648

#define SCHAR_MAX         +127
#define SCHAR_MIN         -128
#define SHRT_MAX          +32767
#define SHRT_MIN          -32768
#define UCHAR_MAX         255U
#define UINT_MAX          65535U
#define ULONG_MAX         4294967295U
#define USHRT_MAX         65535U

#define SINT_MAX          +32767
#define SINT_MIN          -32768
#define SSHRT_MAX         +32767
#define SSHRT_MIN         -32768
```

However, when the **-QU** option, which regards unqualified char as unsigned char, is specified, **CHAR_MAX** and **CHAR_MIN** are declared by the macro **_ _CHAR_UNSIGNED_ _** declared by the compiler as follows.

```
#define CHAR_MAX    (255U)
#define CHAR_MIN    (0)
```

When the **-ZI** option (int and short types are regarded as char type, unsigned int and unsigned short as unsigned char) is specified as a compiler option, **INT_MAX**, **INT_MIN**, **SHRT_MAX**, **SHRT_MIN**, **SINT_MAX**, **SINT_MIN**, **SSHRT_MAX**, **SSHRT_MIN**, **UINT_MAX**, and **USHRT_MAX** are declared as follows, via the macro **_ _FROM_INT_TO_CHAR_ _** declared by the compiler.

```
#define INT_MAX      CHAR_MAX
#define INT_MIN      CHAR_MIN
#define SHRT_MAX     CHAR_MAX
#define SHRT_MIN     CHAR_MIN
#define SINT_MAX     SCHAR_MAX
#define SINT_MIN     SCHAR_MIN
#define SSHRT_MAX    SCHAR_MAX
#define SSHRT_MIN    SCHAR_MIN
#define UINT_MAX     UCHAR_MAX
#define USHRT_MAX    UCHAR_MIN
```

When the **-ZL** option (long type is regarded as int type and unsigned long as unsigned int) is specified as a compiler option, **LONG_MAX**, **LONG_MIN**, and **ULONG_MAX** are declared as follows, via the macro **_ _FROM_LONG_TO_INT_ _** declared by the compiler.

```
#define LONG_MAX     (+32767)
#define LONG_MIN     (-32768)
#define ULONG_MAX    (65535U)
```

(10) stddef.h

In this header, the following objects have been declared and defined:

[Declaration of **int** type **ptrdiff_t**]

```
typedef int ptrdiff_t;
```

[Declaration of **unsigned int** type **size_t**]

```
typedef unsigned int size_t;
```

[Definition of macro name **NULL**]

```
#define NULL (void*)0;
```

[Definition of macro name **offsetof**]

```
#define offsetof(type, member) ((size_t)&(((type*)0)->member))
```

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

offsetof is expanded to the general integer constant expression that has type **size_t** and the value is an **offset** value in byte units from the start of the structure (that is specified by the type) to the structure member (that is specified by the member specifier).

The member specifier must be the one that the result of evaluation of expression& (t. member specifier) becomes an address constant when **static** type **t;** is declared. When the specified member is a bit field, the operation will not be guaranteed.

(11) math.h (normal model only)

math.h defines the following functions.

Table 10-10 Contents of math.h (1/2)

Function Name \ Existence of -ZI, or -ZL Specification	Normal Model			
	None	ZI	ZL	ZI ZL
acos	√	√	√	√
asin	√	√	√	√
atan	√	√	√	√
atan2	√	√	√	√
cos	√	√	√	√
sin	√	√	√	√
tan	√	√	√	√
cosh	√	√	√	√
sinh	√	√	√	√
tanh	√	√	√	√
exp	√	√	√	√
frexp	√	√	√	√
ldexp	√	√	√	√
log	√	√	√	√
log10	√	√	√	√
modf	√	√	√	√
pow	√	√	√	√
sqrt	√	√	√	√
ceil	√	√	√	√
fabs	√	√	√	√
floor	√	√	√	√
fmod	√	√	√	√
matherr	√	—	√	—
acosf	√	√	√	√
asinf	√	√	√	√
atanf	√	√	√	√
atan2f	√	√	√	√
cosf	√	√	√	√
sinf	√	√	√	√
tanf	√	√	√	√
coshf	√	√	√	√
sinhf	√	√	√	√
tanhf	√	√	√	√
expf	√	√	√	√
frexpf	√	√	√	√
ldexpf	√	√	√	√
logf	√	√	√	√
log10f	√	√	√	√
modff	√	√	√	√

√: Supported

—: Not supported

Table 10-10 Contents of math.h (2/2)

Function Name \ Existence of -ZI, or -ZL Specification	Normal Model			
	None	ZI	ZL	ZI ZL
powf	√	√	√	√
sqrtf	√	√	√	√
ceilf	√	√	√	√
fabsf	√	√	√	√
floorf	√	√	√	√
fmodf	√	√	√	√

√: Supported

The following objects are defined.

[Definition of macro name **HUGE_VAL**]

```
#define HUGE_VAL DBL_MAX
```

(12) float.h

float.h defines the following objects.

When the size of a double type is 32 bits, the macro to be defined are sorted by the macro **_ _DOUBLE_IS_32BITS_ _** declared by the compiler.

```
#ifndef _FLOAT_H

#define FLT_ROUNDS              1
#define FLT_RADIX               2

#ifdef _ _DOUBLE_IS_32BITS_ _
#define FLT_MANT_DIG            24
#define DBL_MANT_DIG            24
#define LDBL_MANT_DIG           24

#define FLT_DIG                 6
#define DBL_DIG                 6
#define LDBL_DIG                6

#define FLT_MIN_EXP             -125
#define DBL_MIN_EXP             -125
#define LDBL_MIN_EXP            -125

#define FLT_MIN_10_EXP          -37
#define DBL_MIN_10_EXP          -37
#define LDBL_MIN_10_EXP         -37

#define FLT_MAX_EXP             +128
#define DBL_MAX_EXP             +128
#define LDBL_MAX_EXP            +128

#define FLT_MAX_10_EXP          +38
#define DBL_MAX_10_EXP          +38
#define LDBL_MAX_10_EXP         +38

#define FLT_MAX                 3.40282347E+38F
#define DBL_MAX                 3.40282347E+38F
#define LDBL_MAX                3.40282347E+38F

#define FLT_EPSILON             1.19209290E-07F
#define DBL_EPSILON             1.19209290E-07F
#define LDBL_EPSILON            1.19209290E-07F

#define FLT_MIN                 1.1749435E-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 (normal model only)

Table 10-11 Contents of assert.h

Existence of -ZI, or -ZL Specification / Function Name	Normal Model			
	None	ZI	ZL	ZI ZL
_ _assertfail	√	√	√	√

√: Supported

assert.h defines the following objects.

```
#ifdef NDEBUG
#define assert(p) ((void)0)
#else
extern int _ _assertfail(char*_ _msg, char*_ _cond, char*_ _file, int_ _line);
#define assert(p) ((p) ? (void)0 : (void)_ _assertfail(
    "Assertion failed:  %s, file %s, line %d\n", #p, _ _FILE_ _, _ _LINE_ _))
#endif /* NDEBUG */
```

However, if the **assert.h** header file references another macro, NDEBUG, which is not defined by the **assert.h** header file, and if NDEBUG is defined as a macro when the **assert.h** is captured to the source file, the **assert.h** header file simply declares the assert macro as:

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

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

10.3 Re-entrantability (Normal Model Only)

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 the 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 uses the area secured in the start-up routine
div, **ldiv**, **brk**, **sbrk**, **rand**, **srand**, **strtok**

(3) Functions that deals with floating point numbers
sprintf, **sscanf**, **printf**, **scanf**, **vprintf**, **vsprintf**[Note], **atof**, **strtod**, all the mathematical functions

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

10.4 Standard Library Functions

This section explains the standard library functions of this C compiler by classifying them 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 is 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, new-line, vertical tab, and form-feed)
ispunct	Punctuation characters except white-space characters
isprint	Printable characters
isgraph	Printable nonblank characters
iscntrl	Control characters
isascii	ASCII 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 the 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 where a: Lowercase; A: Uppercase

EXPLANATION

_toupper

- The **_toupper** function is similar to **toupper** except that it does not test to see if the argument is a lowercase letter.

_tolower

- The **_tolower** function is similar to **tolower**, except it does not test to see if the argument is an uppercase letter.

2-1 setjmp, longjmp — Program Control Functions

FUNCTION

The program control function **setjmp** saves the environment information (current state of the program) when a call to this function is made.

The program control function **longjmp** restores the environment information saved by **setjmp**.

HEADER

setjmp. h

FUNCTION PROTOTYPE

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env,int val);
```

Function	Arguments	Return Value
setjmp	**env** ... Array to which environment information is to be saved	• 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**, when called directly, saves **saddr** area, **SP**, and the return address of the function that are used as **HL** register or register variables to **env** and returns 0.

longjmp

- The **longjmp** restores the saved environment to **env** (**saddr** area and **SP** that are used as **HL** register or register variables). Program execution continues as if the corresponding **setjmp** returns **val** (however, if **val** is 0, 1 is returned).

3-1 va_start (normal model only), va_starttop (normal model only), va_arg (normal model only), va_end (normal model only) — Special Functions

FUNCTION

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

The **va_starttop** function (macro) is used to set processing of the variable number of arguments.

The **va_arg** function (macro) obtains the value of an argument from a variable argument list.

The **va_end** function (macro) indicates that the end of a variable argument list is reached.

HEADER

stdarg. h

FUNCTION PROTOTYPE

```
void va_start(va_list ap,parmN);
void va_starttop(va_list ap,parmN);
type va_arg(va_list ap,type);
void va_end(va_list ap);
{va_list is defined as typedef by stdarg.h.}
```

Function	Arguments	Return Value
va_start **va_starttop**	**ap** ... Variable to be initialized so as to be used in **va_arg** and **va_end** **parmN** ... The argument before variable argument	None
va_arg	**ap** ... Variable to process an argument list **type**... Type to point the relevant place of variable argument (**type** is a type of variable length; for example, **int** type if described as **va_arg** (va_list ap, int) or **long** type if described as **va_arg** (**va_list** **ap**, **long**))	Normal case ... Value in the relevant place of variable argument If **ap** is a null pointer ... 0
va_end	**ap** .... Variable to process the variable number of arguments	None

va_start (normal model only), va_starttop (normal model only), va_arg (normal model only), va_end (normal model only) — Special Functions

EXPLANATION

va_start

- In the **va_start** macro, its argument **ap** must be a **va_list** type (**char*** type) object.
- A pointer to the next argument of **parmN** is stored in **ap**.
- **parmN** is the name of the last (right-most) parameter specified in the function's prototype.
- If **parmN** has the **register** storage class, proper operation of this function is not guaranteed.

va_starttop

- The first argument cannot be specified for the **va_start** function if -ZO (option supporting the old interface) is not specified because the first argument is passed by a register.
- Use the macro as follows 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 and subsequent arguments.

va_arg

- In the **va_arg** macro, its argument **ap** must be the same as the **va_list** type object initialized with **va_start** (no guarantee for the other normal operation).
- **va_arg** returns value in the relevant place of variable arguments as a type of **type**.
  The relevant place is the first of variable arguments immediately after **va_start** and next proceeded in each **va_arg**.
- If the argument pointer **ap** is a null pointer, the **va_arg** returns 0 (of **type** type).

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 (normal model only) I/O Functions

FUNCTION

The **sprintf** function writes data into a character string (array) according to the format.

HEADER

stdio.h

FUNCTION PROTOTYPE

```
int sprintf(char *s,const char *format,...);
```

Function	Arguments	Return Value
sprintf	**s** ... Pointer to the string into which the output is to be written **format** ... Pointer to the string which indicates format commands **...** ... Zero or more arguments to be converted	Number of characters written in **s** (Terminating null character is not counted.)

EXPLANATION

- If there are fewer actual arguments than the formats, the proper operation is not guaranteed. In the case that the formats are run out despite the actual arguments still remain, the excess actual arguments are only evaluated and ignored.
- **sprintf** converts zero or more arguments that follow **format** according to the format command specified by **format** and writes (copies) them into the string **s**.
- Zero or more format commands may be used. Ordinary characters (other than format commands that begin with a % character) are output as is to the string **s**. Each format command takes zero or more arguments that follow **format** and outputs them to the string **s**.
- Each format command begins with a % character and is followed by these:
  - Zero or more flags (to be explained later) that modify the meaning of the format command
  - Optional decimal integer which specify a minimum field width

  If the output width after the conversion is less than this minimum field width, this specifier pads the output with blanks of zeros on its left. (If the left-justifying flag "-" (minus) sign follows %, zeros are padded out to the right of the output.)

  The default padding is done with spaces. If the output is to be padded with 0s, place a 0 before the field width specifier. If the number or string is greater than the minimum field width, it will be printed in full even by overrunning the minimum.

sprintf (normal model only) I/O Functions

- Optional precision (number of decimal places) specification (. integer)
  With **d, i, o, u, x,** and **X** type specifiers, the minimum number of digits is specified. With **s** type specifier, the maximum number of characters (maximum field width) is specified. The number of digits to be output following the decimal point is specified for e, E, and f conversions. The number of maximum effective digits is specified for g and G conversions. This precision specification must be made in the form of (.integers). If the integer part is omitted, 0 is assumed to have been specified. The amount of padding resulting from this precision specification takes precedence over the padding by the field width specification.

- Optional **h**, **l** and **L** modifiers
  The **h** modifier instructs the **sprintf** function to perform the **d, i, o, u, x,** or **X** type conversion that follows this modifier on **short int** or **unsigned short int** type. The **h** modifier instructs the sprintf function to perform the **n** type conversion that follows this modifier on a pointer to **short int** type.
  The l modifier instructs the **sprintf** function to perform the **d, i, o, u, x,** or **X** type conversion that follows this modifier on **long int** or **unsigned long int** type. The **h** modifier instructs the **sprintf** function to perform the n type conversion that follows this modifier on a pointer to **long int** type.
  For other type specifiers, the **h**, **l** or **L** modifier is ignored.

- Character that specifies the conversion (to be explained later)
  In the minimum field width or precision (number of decimal places) specification, * may be used in place of an integer string. In this case, the integer value will be given by the **int** argument (before the argument to be converted). Any negative field width resulting from this will be interpreted as a positive field that follows the - (minus) flag. All negative precision will be ignored.

The following flags are used to modify a format command:

Flag	Contents
–	The result of a conversion is left-justified within the field.
+	The result of a signed conversion always begins with a + or - sign.
space	If the result of a signed conversion has no sign, space is prefixed to the output. If the + (plus) flag and space flag are specified at the same time, the space flag will be ignored.
#	The result is converted in the assignment form. In the **o** type conversion, precision is increased so that the first digit becomes 0. In the **x** or **X** type conversion, 0x or 0X is prefixed to a nonzero result. In the e, E, and f type conversions, a decimal point is forcibly inserted to all the output values (in the default without #, a decimal point is displayed only when there is a value to follow). In the g and G type conversions, a decimal point is forcibly inserted to all the output values, and truncation of 0 to follow will not be allowed (in the default without #, a decimal point is displayed only when there is a value to follow. The 0 to follow will be truncated). In all the other conversions, the # flag is ignored.

sprintf (normal model only) I/O Functions

The format codes for output conversion specifications are as follows:

Format code	Contents
d	Converts int argument to signed decimal format.
i	Converts int argument to signed decimal format.
o	Converts int argument to unsigned octal format.
u	Converts int argument to unsigned decimal format.
x	Converts int argument to unsigned hexadecimal format (with lowercase letters abcdef).
X	Converts int argument to unsigned hexadecimal format (with uppercase letters ABCDEF).

With **d**, **i**, **o**, **u**, **x** and **X** type specifiers, the minimum number of digits (minimum field width) of the result is specified. If the output is shorter than the minimum field width, it is padded with zeros. If no precision is specified, 1 is assumed to have been specified. Nothing will appear if 0 is converted with 0 precision.

Precision code	Contents
f	Converts double argument as a signed value with [-] dddd.dddd format. dddd is one or more decimal number(s). The number of digits before the decimal point is determined by the absolute value of the number, and the number of digits after the decimal point is determined by the required precision. When the precision is omitted, it is interpreted as 6.
e	Converts double argument as a signed value with [-] d.dddd e [sign] ddd format. d is 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.

sprintf (normal model only) I/O Functions

Precision code	Contents
p	The associated argument is a pointer to **void** and the pointer value is displayed in hexadecimal 4 digits (with 0s prefixed to less than a 4-digit pointer value). The precision specification if any will be ignored.
n	The associated argument is an integer pointer into which the number of characters written thus far in the string “s” is placed. No conversion is performed.
%	Prints a % sign. The associated argument is not converted (but the flag and minimum field width specifications are effective).

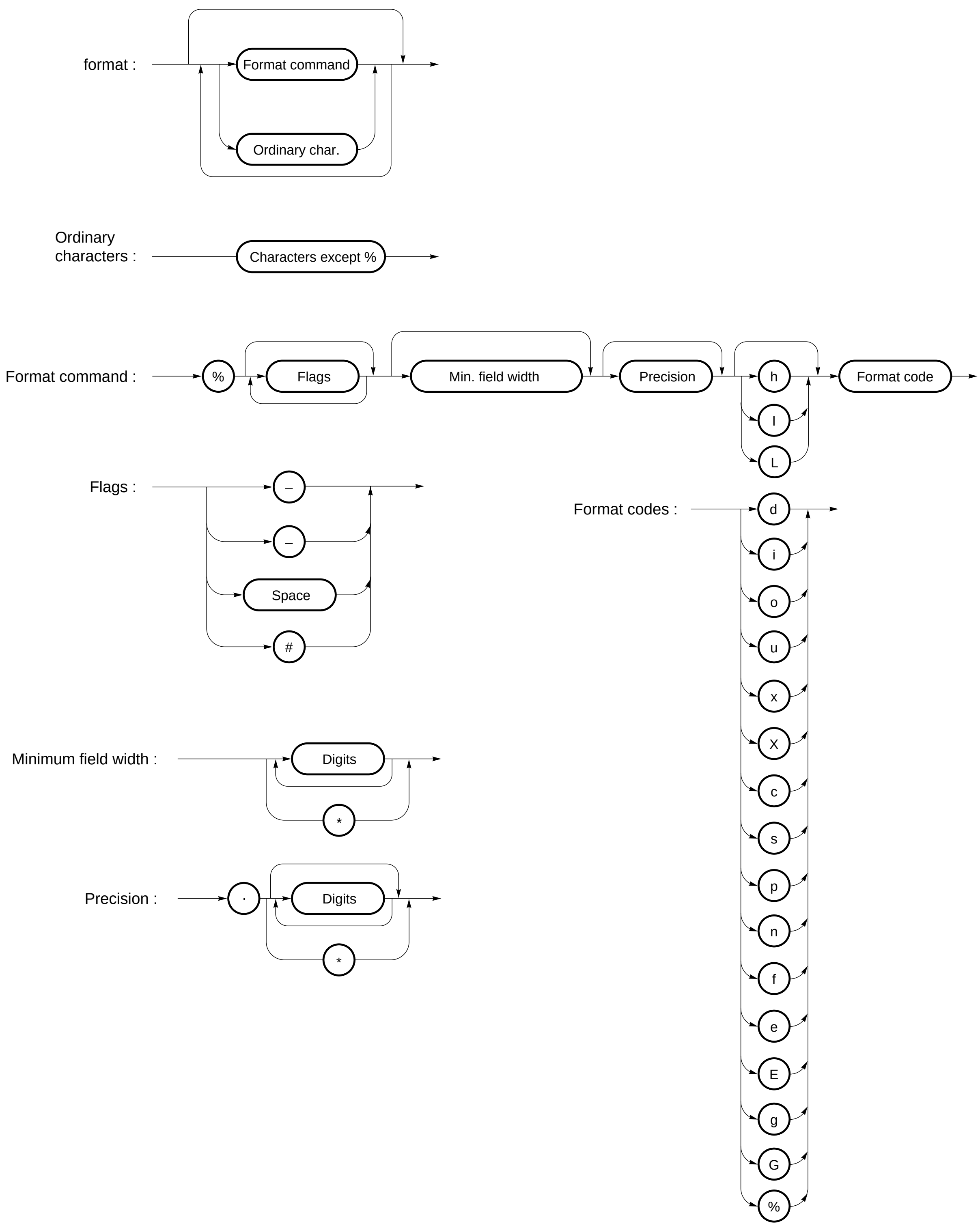
- Operations for invalid conversion specifiers are not guaranteed.
- When the actual argument is a union or a structure, or the pointer to indicate them (except the character type array in % s conversion or the pointer in % p conversion), operations are not guaranteed.
- The conversion result will not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

  non-numeric → “(NaN)”
  +∞ → “(+INF)”
  –∞ →“(–INF)”

sprintf writes a null character at the end of the string **s**. (This character is included in the return value count.)
The syntax of **format** commands is illustrated in **Figure 10-2**.

sprintf (normal model only) I/O Functions

Figure 10-2 Syntax of Format Commands



4-2 sscanf (normal model only) I/O Functions

FUNCTION

The **sscanf** function reads data from the input string (array) according to the format.

HEADER

stdio.h

FUNCTION PROTOTYPE

```
int sscanf(const char *s,const char *format,...);
```

Function	Arguments	Return Value
sscanf	**s** ... Pointer to the input string **format** ... Pointer to the string which indicates the input format commands **...** ... Pointer to object in which converted values are to be stored, and zero or more arguments	–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 which are classified into the following three types:
  (a) White-space characters (one or more characters for which **isspace** becomes true)
  (b) Non-white-space characters (other than %)
  (c) 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 mentioned later)

sscanf (normal model only) I/O Functions

sscanf executes the format commands in "format" in sequence and if any format command fails, the function will terminate.

(a) A white-space character in the control string causes **sscanf** to read any number (including zero) of white-space character up to the first non-white-space character (which will not be read). This white-space character command fails if it does not encounter any non-white-space character.

(b) A non-white-space character causes **sscanf** to read and discard a matching character. This command fails if the specified character is not found.

(c) 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 with the specified type, the command execution fails. Unless assignment is suppressed by *, the result of the conversion is stored in the object pointed to by the first argument which follows "format" and has not yet received the result of the conversion.

The following type specifiers are available:

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

i .... Converts an integer (which may be signed). If a number is preceded by 0x or 0X, the number is interpreted as a hexadecimal integer. If a number is preceded by 0, the number is interpreted as an octal integer. Other numbers are regarded as decimal integers. The corresponding argument must be a pointer to an integer.

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

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

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

e, **E**, **f**, **g**, **G**........... Floating point value consists of optional sign (+ or –), one or more consecutive decimal number(s) including decimal point, optional exponent (**e** or **E**), and the following optional signed integer value. When overflow occurs as a result of conversion, or when underflow occurs with the conversion result ±∞, non-normalized number or ±0 becomes the conversion result. The corresponding argument is a pointer to float.

sscanf (normal model only) I/O Functions

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

[........................... Inputs a character string consisting of expected character groups (called a **scanset**). The corresponding argument must be a pointer to the first character of an array that has sufficient size to accommodate this character string and a null terminator. The null terminator will be automatically added. The format commands continue from this character up to the closing square bracket (1). The character string (called a **scanlist**) enclosed in the square brackets constitutes a **scanset** except when the character immediately after the opening square bracket is a circumflex ( ).
When the character is a circumflex, all the characters other than a **scanlist** between the circumflex and the closing square bracket constitute a **scanset**. However, when a **scanlist** begins with [ ] or [^], this closing square bracket is contained in the **scanlist** and the next closing square list becomes the end of the **scanlist**.
A hyphen (–) at other than the left or right end of a **scanlist** is interpreted as the punctuation mark for hyphenation if the character at the left of the range specifying hyphen (–) is not smaller than the right-hand character in ASCII code value.

c .... Inputs a character string consisting of the number of characters specified by the field width. (If the field width specification is omitted, 1 is assumed.) The corresponding argument must be a pointer to the first character of an array that has sufficient size to accommodate this character string. The null terminator will not be added.

p .... Reads an unsigned hexadecimal integer. The corresponding argument must be a pointer to **void** pointer.

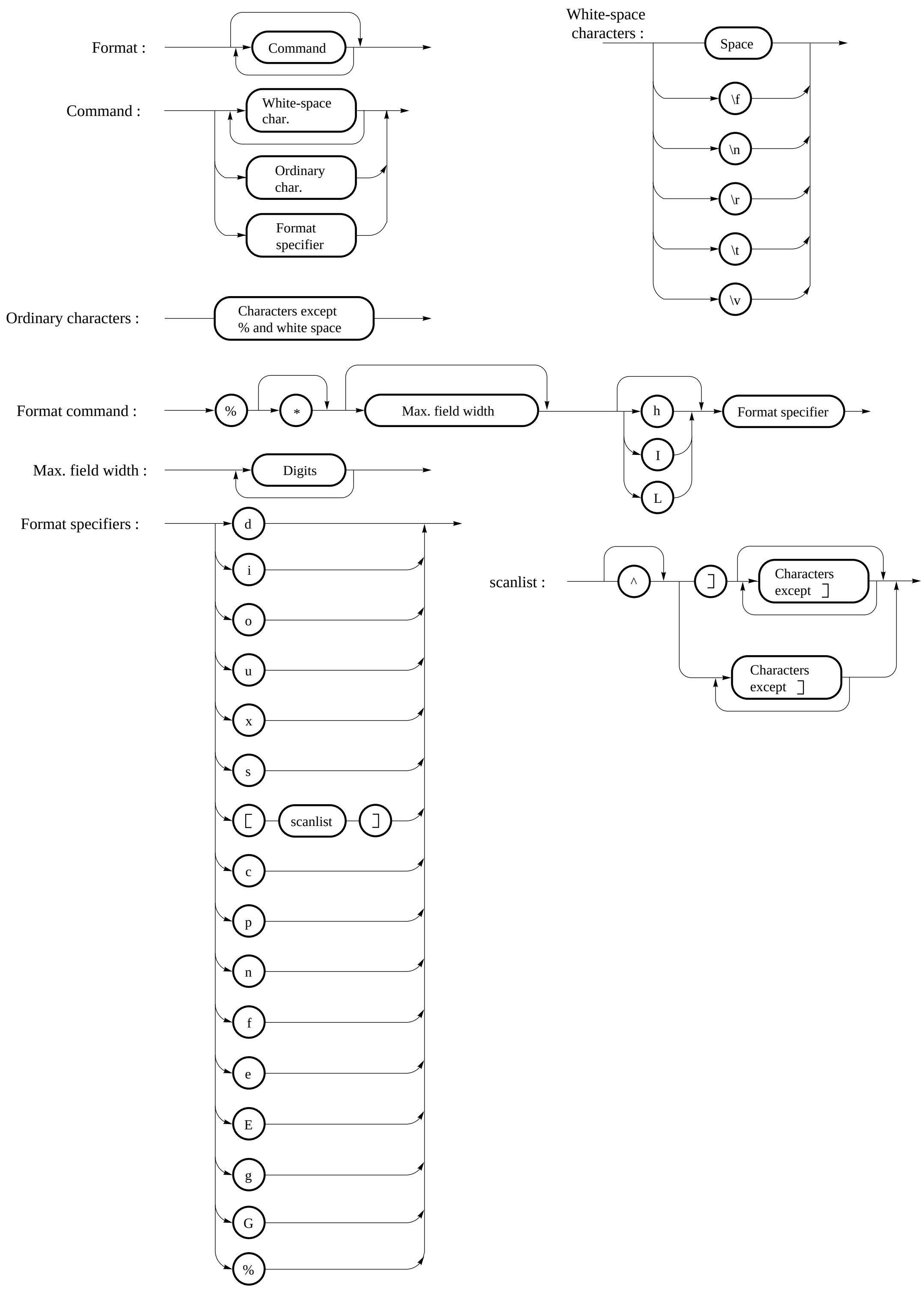
n .... Receives no input from the string **s**. The corresponding argument must be a pointer to an integer. The number of characters that are read thus far by this function from the string “s” is stored in the object that is pointed to by this pointer. The %n format command is not included in the return value assignment count.

% .... Reads a % sign. Neither conversion nor assignment takes place.

If a format specification is invalid, the format command execution fails.
If a null terminator appears in the input stream, **sscanf** will terminate.
If an overflow occurs in an integer conversion (with the **d, i, o, u, x,** or **p** format specifier), high-order bits will be truncated depending on the number of bits of the data type after the conversion.
The syntax of input **format** commands is illustrated below.

sscanf (normal model only) I/O Functions

Figure 10-3 Syntax of Input Format Commands



4-3 printf (normal model only) 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 character output to s (the null character at the end is not counted)

EXPLANATION

- (0 or more) arguments following the format are converted and output using the putchar function, according to the output conversion specification specified in the format.
- The output conversion specification is 0 or more directives. Normal characters (other than the conversion specification starting with %) are output as is using the putchar function. The conversion specification is output using the **putchar** function by fetching and converting the following (0 or more) arguments.
- Each conversion specification is the same as that of the **sprintf** function.

4-4 scanf (normal model only) 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 **...** ... Pointer (0 or more) argument to the object to assign the converted value	When the character string **s** is not null ... number of input items assigned

EXPLANATION

- Performs input using **getchar** function. Specifies input string permitted by the character string **format** indicates. Uses the argument after the format as the pointer to an object. **format** specifies how the conversion is performed by the input string.
- When there are not enough arguments for the **format**, normal operation is not guaranteed. When the argument is excessive, the expression will be evaluated but not input.
- **format** consists of 0 or more directives. The directives are as follows.

  (1) One or more null character (character that makes isspace true)
  (2) Normal character (other than %)
  (3) Conversion indication

- If a conversion ends with a input character which conflicts with the input character, the conflicting input character is rounded down. The conversion indication is the same as that of the **sscanf** function.

4-5 vprintf (normal model only) 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 **putchar** function according to the output conversion specification specified by the format.
- Each conversion specification is the same as that of **sprintf** function.

4-6 vsprintf (normal model only) 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 which s indicates according to the output conversion specification specified by **format**.
- The output specification is the same as that of **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).
- Error check related to reading is not performed.
- To change SFR to read, it is necessary either that the source be changed to be re-registered to the library or that the user create a new **getchar** function.

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 remains 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).
- Error check related to writing is not performed.
- To change SFR to write, it is necessary either that the source is changed and re-registered to the library or the user create a new **putchar** function.

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 **putchar** function, a line feed character is added at the end of the output.
- Writing of the null character at the end of the character string is not performed.
- When the return value is normal, 0 is returned, and when **putchar** function returns –1, –1 is returned.

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 int** value.

HEADER

stdlib.h

FUNCTION PROTOTYPE

```
int atoi(const char *nptr);
long int atol(const char *nptr);
```

Function	Arguments	Return Value
atoi	**nptr**... string to be converted	• **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 **atoi** function skips over zero or more white-space characters (for which **isspace** becomes true) from the beginning of the string and converts the string from the character next to the skipped white-spaces to an integer (until other than digits or a null character appears in the string). If no digits to convert is found in the string, the function returns 0. If an overflow occurs, the function returns **INT_MAX** (32767) for positive overflow and **INT_MIN** (-32768) for negative overflow.

atol

- The **atol** function converts the first part of the string pointed to by pointer **nptr** to a **long int** value.
- The **atol** function skips over zero or more white-space characters (for which **isspace** becomes true) from the beginning of the string and converts the string from the character next to the skipped white-spaces to an integer (until other than digits or null character appears in the string). If no digits to convert is found in the string, the function returns 0. If an overflow occurs, the function returns **LONG_MAX** (2147483647) for positive overflow and **LONG_MIN** (–2147483648) for 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);
```

<table>
<tr><th>Function</th><th>Arguments</th><th>Return Value</th></tr>
<tr><td>strtol</td><td rowspan="2">nptr... string to be converted
endptr ... Address of char pointer
base ... base for number represented in the string</td><td>• long int value if converted properly
• LONG_MAX (2147483647) for positive overflow
• LONG_MIN (–2147483648) for negative overflow
• 0 if not converted</td></tr>
<tr><td>strtoul</td><td>• unsigned long if converted properly
• ULONG_MAX (4294967295U) if overflow occurs
• 0 if not converted</td></tr>
</table>

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 the part (2) of the string into an integer and returns this integer value.

- A **base** of 0 indicates that the **base** should be determined from the leading digits of the string. A leading 0x or 0X indicates a hexadecimal number; a leading 0 indicates an octal number; otherwise, the number is interpreted as decimal. (In this case, the number may be signed).
- If the **base** is 2 to 36, the set of letters from a to z or A to Z which can be part of a number (and which may be signed) with any of these bases are taken to represent 10 to 35. A leading 0x or 0X is ignored if the base is 16.
- If **endptr** is not a null pointer, a pointer to the part (3) of the string is stored in the object pointed to by **endptr**.
- If the correct value causes an overflow, the function returns **LONG_MAX** (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 (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 the part (2) of the string into a unsigned integer and returns this unsigned integer value.

- A **base** of 0 indicates that the **base** should be determined from the leading digits of the string. A leading 0x or 0X indicates a hexadecimal number; a leading 0 indicates an octal number; otherwise, the number is interpreted as decimal.
- If the **base** is 2 to 36, the set of letters from a to z or A to Z which can be part of a number (and which may be signed) with any of these bases are taken to represent 10 to 35. A leading 0x or 0X is ignored if the **base** is 16.
- If **endptr** is not a null pointer, a pointer to the part (3) of the string is stored in the object pointed to by **endptr**.

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 (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.

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.
- Returns the pointer to the beginning of the allocated area if the requested size is allocated.
- Returns the null pointer if the requested size is not allocated.
- The memory allocation will start from a break value and the address next to the allocated space will become a new break value. See **10.4 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**. (The **malloc, calloc,** or **realloc** called after the **free** will give you the space that was freed earlier.)
- If **ptr** does not point to the allocated space, the **free** will take no action. (Freeing the allocated space is performed by setting **ptr** as a new break value.)

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 **10.4 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 warp-up function can be registered, **atexit** returns 0. If no more wrap-up function can be registered because 32 wrap-up functions have already been registered, the function returns 1.

exit

- The **exit** function causes immediate, normal termination of a program.
- This function calls the wrap-up functions in the reverse of the order in which they were registered with **atexit**.
- The **exit** function loops and can never return to its caller.
- The user must create the **exit** processing routine.

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 (normal model only), ldiv (normal model only) — 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 and the remainder to the **rem** element of **div_t** type member
ldiv		Quotient to the **quot** element and the remainder to the **rem** element of **ldiv_t** type member

EXPLANATION

div

- The **div** function performs the integer division of numerator divided by denominator.
- The absolute value of quotient is defined as the largest integer not greater than the absolute value of **numer** divided by the absolute value of **denom**. The remainder always has the same sign as the result of the division (plus if **numer** and **denom** have the same sign; otherwise minus).
- The remainder is the value of **numer – denom***quotient.
- If **denom** is 0, the quotient becomes 0 and the remainder becomes numer.
- If **numer** is –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 absolute value of quotient is defined as the largest long int type integer not greater than the absolute value of numer divided by the absolute value of denom. The remainder always has the same sign as the result of the division (plus if numer and denom have the same sign; otherwise minus).
- The remainder is the value of **numer – denom***quotient.
- If **denom** is 0, the quotient becomes 0 and the remainder becomes **numer**.
- If **numer** is –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 block to be released	• 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

The string function **atof** converts the contents of a decimal integer string to a **double** value.
The string function **strtod** converts the contents of a string to a **double** value.

HEADER

stdlib.h

FUNCTION PROTOTYPE

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

Function	Arguments	Return Value
atof	**nptr** ... string to be converted	• Converted value if converted properly • **HUGE_VAL** (with sign of overflowed value) if positive overflow occurs • 0 if negative overflow occurs • 0 if the string is invalid
strtod	**nptr** ... string to be converted **endptr** … pointer storing pointer pointing to unrecognizable block	• Converted value if converted properly • **HUGE_VAL** (with sign of overflowed value) if positive overflow occurs • 0 if negative overflow occurs • 0 if the string is invalid

atof, strtod — Utility Functions

EXPLANATION

atof

- The **atof** function converts the string pointed to by pointer **nptr** to a **double** value.
- The **atof** function skips over zero or more white-space characters (for which **isspace** becomes true) from the beginning of the string and converts the string from the character next to the skipped white-spaces to a floating-point number (until other than digits or a null character appears in the string).
- A floating-point number is returned when converted properly.
- If an overflow occurs on conversion, HUGE_VAL with the sign of the overflowed value is returned and ERANGE is set to errno.
- If valid digits are deleted due to an underflow or an overflow, a non-normalized number and ±0 are returned respectively, and ERANGE is set to errno.
- IF conversion cannot be performed, 0 is returned.

strtod

- The **strtod** function converts the string pointed to by pointer **nptr** to a **double** value.
- The **strtod** function skips over zero or more white-space characters (for which **isspace** becomes true) from the beginning of the string and converts the string from the character next to the skipped white-spaces to a floating-point number (until other than digits or null character appears in the string).
- A floating-point number is returned when converted properly.
- If an overflow occurs on conversion, HUGE_VAL with the sign of the overflowed value is returned and ERANGE is set to errno.
- If valid digits are deleted due to an underflow or an overflow, a non-normalized number and ±0 are returned respectively, and ERANGE is set to errno. In addition, endptr stores a pointer for next character string at that time.
- IF conversion cannot be performed, 0 is returned.

5-13 itoa, ltoa (normal model only), ultoa (normal model only) — Utility Functions

FUNCTION

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

The string function **ltoa** converts a **long int** integer to its string equivalent.

The string function **ultoa** converts an **unsigned long** integer to its string equivalent.

HEADER

stdlib.h

FUNCTION PROTOTYPE

```
char *itoa(int value,char *string,int radix);
char *ltoa(long value,char *string,int radix);
char *ultoa(unsigned long value,char *string,int radix);
```

Function	Arguments	Return Value
itoa, **ltoa,** **ultoa**	**value** ... String to which integer is to be converted **string** ... Pointer to the conversion result **radix** ... Base of output string	• 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 pseudorandom numbers.

The mathematical function **srand** sets a starting value (seed) for the sequence generated by **rand**.

HEADER

stdlib.h

FUNCTION PROTOTYPE

```
int rand(void);
void srand(unsigned int seed);
```

Function	Arguments	Return Value
rand	None	Pseudorandom integer in the range of 0 to **RAND_MAX**
srand	**seed** ... Starting value for pseudorandom number generator	None

EXPLANATION

rand

- Each time the **rand** function is called, it returns a pseudorandom integer in the range of 0 to **RAND_MAX**.

srand

- The **srand** function sets a starting value for a sequence of random numbers. **seed** is used to set a starting point for a progression of random numbers that is a return value when **rand** is called. If the same **seed** value is used, the sequence of pseudorandom numbers is the same when **srand** is called again.
- Calling **rand** before **srand** is used to set a seed is the same as calling **rand** after **srand** has been called with **seed** = 1. (The default **seed** is 1.)

5-15 bsearch (normal model only) 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 which 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 which consists of **nmemb** number of members each of which has the size specified by **size** and must have been sorted in ascending order.
- The function pointed to by **compare** takes 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 (normal model only) 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 elements 1 and 2), compares the two arguments, and returns:
- The array element 1 as the 1st argument and array element 2 as the 2nd argument

  Negative value if the 1st argument is less than the 2nd argument
  0 if both arguments are equal
  Positive integer if the 1st argument is greater than the 2nd argument

- If the 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	**ends** ... break value to set	Normal ... 0 When a break value cannot be changed ... –1

EXPLANATION

- Sets the value given by **endds** to the break value (the address following the address at the end of the area to be allocated).
- When **endds** is out of the permissible range, the break value is not changed. **ENOMEM**(3) is set to **errno** and –1 is returned.

5-18 strsbrk Utility Functions

FUNCTION

strsbrk increases/decreases a break value.

HEADER

stdlib.h

FUNCTION PROTOTYPE

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

Function	Arguments	Return Value
strsbrk	**incr** ... amount to increase/decrease a break value	Normal ... Old break value When a break value cannot be increased/decreased ... –1

EXPLANATION

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

5-19 stritoa, strltoa (normal model only), strultoa (normal model only) — 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

stdlib.h

FUNCTION PROTOTYPE

```
char *stritoa(int value,char *string,int radix);
char *strltoa(long value,char *string,int radix);
char *strultoa(unsigned long value,char *string,int radix);
```

Function	Arguments	Return Value
stritoa **strltoa** **strultoa**	**value** ... character string to convert **string** ... pointer to conversion result **radix** ... radix to specify	Normal ... pointer to the converted character string Others ... null pointer

EXPLANATION

stritoa, strltoa, strultoa

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

6-1 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**.
- If the string pointed to by **s2** has fewer characters than specified by n, the **strncat** function concatenates the string including the null terminator. If there are more characters than specified by n, the n character section is concatenated starting from the top.
- The null terminator must always be concatenated.
- When copying is performed between objects overlapping each other, the operation is not guaranteed.

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**.

  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 area.

HEADER

string.h

FUNCTION PROTOTYPE

```
int strcoll (const char *s1, const char *s2) ;
```

Function	Arguments	Return Value
strcoll	**s1** ... pointer to comparison character string **s2** ... pointer to comparison character string	When character strings **s1** and **s2** are equal ... 0 When character strings **s1** and **s2** are different ... The difference between the values whose first different characters are converted to int (character of **s1** – character of **s2**)

EXPLANATION

- This compiler does not support operations specific to 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 area.

HEADER

string.h

FUNCTION

```
size_t strxfrm (char *s1, const char *s2, size_t n) ;
```

Function	Arguments	Return Value
strxfrm	**s1** ... pointer to a compared character string **s2** ... pointer to a compared character string **n** ... Maximum number of characters to s1	Returns the length of the character string of the result of the conversion (does not include a character string to indicate the end) If the returned value is n or more, the contents of the array indicated by **s1** is undefined.

EXPLANATION

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

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

  return (**strlen** (**s2**)) ;

7-1 acos (normal model only) 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 ≤ x ≤ 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 (normal model only) 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 ≤ x ≤ 1 ... **asin** of **x** When x ≤ –1, 1 ≤ x, x = NaN ... **NaN** When **x** = –0 ... –0 When underflow occurs ... non-normalized number

EXPLANATION

- Calculates **asin** (range between –π/2 and +π/2) of **x**.
- In the case of area error of **x** < –1, 1 < **x**, **NaN** is returned and **EDOM** is set to **errno**.
- When **x** is non-numeric, **NaN** is returned.
- When **x** is –0, –0 is returned.
- If underflow occurs as a result of conversion, a non-normalized number is returned.

7-3 atan (normal model only) 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 –π/2 and +π/2) of **x**.
- When **x** is non-numeric, **NaN** is returned.
- When **x** is –0, –0 is returned.
- If underflow occurs as a result of conversion, a non-normalized number is returned.

7-4 atan2 (normal model only) 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 ± ∞ ... **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 underflow occurs as a result of operation, non-normalized number is returned.

7-5 cos (normal model only) 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 (normal model only) 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 underflow occurs as a result of operation, a non-normalized number is returned.
- If the absolute value of **x** is extremely large, the result of an operation becomes an almost meaningless value.

7-7 tan (normal model only) 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 underflow occurs as a result of operation, a non-normalized number is returned.
- If the absolute value of **x** is extremely large, the result of an operation becomes an almost meaningless value.

7-8 cosh (normal model only) 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 overflow occurs as a result of operation, **HUGE_VAL** with a positive sign is returned, and **ERANGE** is set to **errno**.

7-9 sinh (normal model only) 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 overflown value) When underflow occurs ... ±0

EXPLANATION

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

7-10 tanh (normal model only) 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 underflow occurs as a result of operation, ±0 is returned.

7-11 exp (normal model only) 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 exponent function of **x**.
- If **x** is non-numeric, NaN is returned.
- If **x** is ±∞, ±∞ is returned.
- If underflow occurs as a result of operation, non-normalized number is returned.
- If annihilation of valid digits due to underflow occurs as a result of operation, +0 is returned.
- If overflow occurs as a result of operation, **HUGE_VAL** with a positive sign is returned and **ERANGE** is set to **errno**.

7-12 frexp (normal model only) Mathematical Functions

FUNCTION

frexp finds mantissa and exponent part.

HEADER

math.h

FUNCTION PROTOTYPE

```
double frexp (double x, int *exp) ;
```

Function	Arguments	Return Value
frexp	**x** ... numeric value to perform operation **exp** ... pointer to store exponent part	Normal ... mantissa of **x** When x = NaN, x = ±∞ ... **NaN** When x = ±0 ... ±0

EXPLANATION

- Divide a floating point number **x** to mantissa **m** and exponent **n** such as **x** = m*2^n and returns mantissa **m**.
- Exponent n is stored where the pointer **exp** indicates. The absolute value of **m**, however, is 0.5 or more and less than 1.0.
- If **x** is non-numeric, **NaN** is returned and the value of ***exp** is 0.
- If **x** is infinite, NaN is returned, and **EDOM** is set to **errno** with the value of ***exp** as 0.
- If **x** is ±0, **±0** is returned and the value of ***exp** is 0.

7-13 ldexp (normal model only) 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 overflown value) When underflow occurs ... Non-normalized number When annihilation of valid digits occurs due to underflow ... ±0

EXPLANATION

- Calculates **x***2^**exp**.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is ±∞, ±∞ is returned.
- If **x** is ±0, ±0 is returned.
- If overflow occurs as a result of operation, **HUGE_VAL** with the overflown value is returned and **ERANGE** is set to **errno**.
- If underflow occurs as a result of operation, non-normalized number is returned.
- If annihilation of valid digits due to underflow occurs as a result of operation, ±0 is returned.

7-14 log (normal model only) Mathematical Functions

FUNCTION

log finds natural logarithm.

HEADER

math.h

FUNCTION PROTOTYPE

```
double log (double x) ;
```

Function	Arguments	Return Value
log	**x** ... numeric value to perform operation	Normal ... Natural logarithm of **x** When x ≤ 0 ... **HUGE_VAL** (with negative sign) When x is non-numeric ... **NaN** When x is infinite ... +∞

EXPLANATION

- Finds natural logarithm of **x**.
- If **x** is non-numeric, **NaN** is returned.
- If **x** is +∞, +∞ is returned.
- In the case of 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 (normal model only) Mathematical Functions

FUNCTION

log10 finds logarithm with 10 as the base.

HEADER

math.h

FUNCTION PROTOTYPE

```
double log10 (double x) ;
```

Function	Arguments	Return Value
log10	**x** ... numeric value to perform operation	Normal ... logarithm with 10 of **x** as the base When x ≤ 0 ... **HUGE_VAL** (with negative sign) When x is non-numeric ... **NaN** When x is infinite ... +∞

EXPLANATION

- Finds 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 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 (normal model only) Mathematical Functions

FUNCTION

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

7-17 pow (normal model only) Mathematical Functions

FUNCTION

pow finds yth power of **x**.

HEADER

math.h

FUNCTION PROTOTYPE

```
double pow (double x, double y) ;
```

Function	Arguments	Return Value
pow	**x** ... numeric value to perform operation **y** ... multiplier	Normal ... **x^y** Either when x = NaN or y = NaN, x = +∞ and y = 0 x < 0 and y≠ integer, x < 0 and y = ±∞, x = 0 and y ≤ 0 ... NaN When underflow occurs ... Non-normalized number When overflow occurs ... **HUGE_VAL** (with the sign of overflown value) When annihilation of valid digits occurs due to underflow ... ±0

EXPLANATION

- Calculates **x^y**.
- If overflow occurs as a result of 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 underflow occurs, a non-normalized number is returned.
- If annihilation of valid digits occurs due to underflow, ±0 is returned.

7-18 sqrt (normal model only) Mathematical Functions

FUNCTION

sqrt finds square root.

HEADER

math.h

FUNCTION PROTOTYPE

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

Function	Arguments	Return Value
sqrt	**x** ... numeric value to perform operation	When x ≥ 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.

7-19 ceil (normal model only) 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 (normal model only) 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 (normal model only) 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 (normal model only) 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 (normal model only) 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 library and run-time library that deal with floating-point numbers.
- When called from the standard library, **EDOM** and **ERANGE** are set to **errno**.
  The following shows the relationship between the arithmetic exception type and **errno**.

Type	Arithmetic Exception	Value Set to **errno**
1	Underflow	**ERANGE**
2	Annihilation	**ERANGE**
3	Overflow	**ERANGE**
4	Zero division	**EDOM**
5	Inoperable	**EDOM**

Original error processing can be performed by changing or creating matherr.

7-24 acosf (normal model only) 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 $-1 \le x \le 1$ ... acos of **x** When $x \le -1$, $1 < x$, x = ... **NaN**

EXPLANATION

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

7-25 asinf (normal model only) 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 ≤ x ≤ 1 ... asin of **x** When x ≤ –1, 1 < x, x = NaN ... **NaN** x = –0 ... –0 When underflow occurs ... Non-normalized number

EXPLANATION

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

7-26 atanf (normal model only) 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 underflow occurs as a result of operation, a non-normalized number is returned.

7-27 atan2f (normal model only) Mathematical Functions

FUNCTION

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

HEADER

math.h

FUNCTION PROTOTYPE

```
float atan21 (float y, float x) ;
```

Function	Arguments	Return Value
atan21	**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 underflow occurs as a result of operation, a non-normalized number is returned.

7-28 cosf (normal model only) 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 (normal model only) 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 underflow occurs as a result of operation, a non-normalized number is returned.
- If the absolute value of **x** is extremely large, the result of an operation becomes an almost meaningless value.

7-30 tanf (normal model only) 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 underflow occurs as a result of operation, a non-normalized number is returned.
- If the absolute value of **x** is extremely large, the result of an operation becomes an almost meaningless value.

7-31 coshf (normal model only) 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 overflow occurs as a result of operation, **HUGE_VAL** with a positive sign is returned and **ERANGE** is set to **errno**.

7-32 sinhf (normal model only) 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 overflown 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 overflow occurs as a result of operation, **HUGE_VAL** with the sign of overflown value is returned and **ERANGE** is set to errno.
- If underflow occurs as a result of operation, ±0 is returned.

7-33 tanhf (normal model only) 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 underflow occurs as a result of operation, ±0 is returned.

7-34 expf (normal model only) Mathematical Functions

FUNCTION

expf finds **exponent** function.

HEADER

math.h

FUNCTION PROTOTYPE

```
float expf (float x) ;
```

Function	Arguments	Return Value
expf	**x** ... numeric value to perform operation	Normal ... exponent function of **x** When overflow occurs ... **HUGE_VAL** (with positive sign) x = NaN ... **NaN** When x = ±∞ ... ±∞ When underflow occurs ... Non-normalized number When annihilation of effective digits occurs due to underflow ... +0

EXPLANATION

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

7-35 frexpf (normal model only) Mathematical Functions

FUNCTION

frexpf finds mantissa and exponent part.

HEADER

math.h

FUNCTION PROTOTYPE

```
float frexpf (float x, int *exp) ;
```

Function	Arguments	Return Value
frexpf	**x** ... numeric value to perform operation **exp** ... pointer to store exponent part	Normal ... mantissa of **x** When x = NaN, x = ±∞ ... **NaN** When x = ±0 ... ±0

EXPLANATION

- Divides a floating-point number **x** to mantissa m and exponent n such as **x** = **m***2^**n** and returns mantissa m.
- Exponent **n** is stored in where the pointer **exp** indicates. The absolute value of **m**, however, is 0.5 or more and less than 1.0.
- If **x** is non-numeric, **NaN** is returned and the value of ***exp** is 0.
- If **x** is ±∞, NaN is returned, and **EDOM** is set to errno with the value of ***exp** as 0.
- If **x** is ±0, ±0 is returned and the value of ***exp** is 0.

7-36 ldexpf (normal model only) 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 overflown 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 overflown value is returned and **ERANGE** is set to **errno**.
- If underflow occurs as a result of operation, non-normalized number is returned .
- If annihilation of valid digits due to underflow occurs as a result of operation, ±0 is returned.

7-37 logf (normal model only) Mathematical Functions

FUNCTION

logf finds natural logarithm.

HEADER

math.h

FUNCTION PROTOTYPE

```
float logf (float x) ;
```

Function	Arguments	Return Value
logf	**x** ... numeric value to perform operation	Normal ... Natural logarithm of **x** When x 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 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 (normal model only) Mathematical Functions

FUNCTION

log10f finds logarithm with 10 as the base.

HEADER

math.h

FUNCTION PROTOTYPE

```
float log10f (float x) ;
```

Function	Arguments	Return Value
log10f	**x** ... numeric value to perform operation	Normal ... logarithm with 10 of **x** as the base When x is non-numeric ... **NaN** When x = +∞ ... +∞ When x ≤ 0 ... **HUGE_VAL** (with negative sign)

EXPLANATION

- Finds 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 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 (normal model only) Mathematical Functions

FUNCTION

modff finds fraction part and integer part.

HEADER

math.h

FUNCTION PROTOTYPE

```
float modff (float x, float *iptr) ;
```

Function	Arguments	Return Value
modff	**x** ... numeric value to perform operation **iptr** ... Pointer for integer part	Normal ... fraction part of **x** When x is non-numeric or infinite ... **NaN** When x = ±0 ... ±0

EXPLANATION

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

7-40 powf (normal model only) Mathematical Functions

FUNCTION

powf finds yth power of **x**.

HEADER

math.h

FUNCTION PROTOTYPE

```
float powf (float x, float y) ;
```

Function	Arguments	Return Value
powf	**x** ... numeric value to perform operation **y** ... multiplier	Normal ... **x^y** Either when = x = NaN or y = **NaN** x = +∞ and y = 0 x < 0 and y≠ integer, x < 0 and y = ±∞ x = 0 and y ≤ 0 ... NaN When underflow occurs ... Non-normalized number When overflow occurs ... **HUGE_VAL** (with the sign of overflown value) When annihilation of valid digits occurs due to underflow ... ±0

EXPLANATION

- Calculates **x^y**.
- If overflow occurs as a result of 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 underflow occurs, a non-normalized number is returned.
- If annihilation of valid digits occurs due to underflow, ±0 is returned.

7-41 sqrtf (normal model only) Mathematical Functions

FUNCTION

sqrtf finds square root.

HEADER

math.h

FUNCTION PROTOTYPE

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

Function	Arguments	Return Value
sqrtf	**x** ... numeric value to perform operation	When x ≥ 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 (normal model only) 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 (normal model only) 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 (normal model only) 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 (normal model only) 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 (normal model only) Diagnostic Functions

FUNCTION

_ _assertfail supports **assert** macro.

HEADER

assert.h

FUNCTION PROTOTYPE

```
int _ _assertfail (char*_ _msg, char*_ _cond, char*_ _file, int_ _line) ;
```

Function	Arguments	Return Value
_ _assertfail	**_ _msg** ... pointer to character string to indicate output conversion specification to be passed to **printf** function **_ _cond** ... actual argument of **assert** macro **_ _file** ... source file name **_ _line** ... source line number	Undefined

EXPLANATION

- A **_ _assertfail** function receives information from assert macro (refer to **10.2 (13) assert.h**), calls **printf** function, outputs information, and calls **abort** function.
- An **assert** macro adds diagnostic function to a program. When an assert macro is executed, if **p** is false (equal to 0), an **assert** macro passes information related to the specific call that has brought the false value (actual argument text, source file name, and source line number are included in the information. The other two are the values of macro **_ _FILE_ _** and **_ _LINE_ _**, respectively) to **_ _assertfail** function.

10.5 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 10-12** below.

Caution The files d002.78k and d014.78k in the BAT directory are used during batch file activation for updating library, not for development. When developing a system, it is necessary to have a device file (sold separately).

Table 10-12 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 SM78K0-supporting. When it is necessary to check the output of the **putchar** function using the SM78K0, 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.
repbank.bat	Updates the bank function call processing routine (bankcall.asm). The default assumption sets P0 of the SFR to the bank switching control port. When it is necessary to change this setting, change the defined value of EQU of PORT in bankcall.asm and update the library using this batch file.
repvect.bat	Updates the address value setting processing of the branch table of the interrupt vector table allocated in the flash area (vect*.asm). The default assumption sets the top address of the flash area branch table to 2000H. When it is necessary to change this setting, change the defined value of EQU of ITBLTOP in vect.inc and update the library using this batch file.

10.5.1 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 RA78K0 assembler package Ver. 3.50 or later operates is necessary. Before using the batch files, set the directory that contains the RA78K0 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)
054 ··· μPD78054, etc.

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

The batch file for:

(1) Startup routine

- For IBM PC/AT and compatibles
  mkstup chiptype

Example `mkstup 054`

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

Example `/bin/sh mkstup.sh 054`

(2) Firmware ROM routine update

- For IBM PC/AT and compatibles
  reprom chiptype multiply/divide instruction existence

Example `reprom 054 use`

- For HP9000 series 700, SPARCstation Family
  /bin/sh reprom.sh chiptype multiply/divide instruction existence

Example `/bin/sh reprom.sh 054 use`

(3) Bank function call processing routine update

- For IBM PC/AT and compatibles
  repbank chiptype multiply/divide instruction existence

Example `repbank 054 use`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repbank.sh chiptype multiply/divide instruction existence

Example `/bin/sh repbank.sh 054 use`

(4) getchar function update

- For IBM PC/AT and compatibles
  repgetc chiptype multiply/divide instruction existence

Example `repgetc 054 use`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repgetc.sh chiptype multiply/divide instruction existence

Example `/bin/sh repgetc.sh 054 use`

(5) putchar function update

- For IBM PC/AT and compatibles
  repputc chiptype multiply/divide instruction existence

Example `repputc 054 use`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repputc.sh chiptype multiply/divide instruction existence

Example `/bin/sh repputc.sh 054 use`

(6) putchar function (SM78K0-supporting) update

- For IBM PC/AT and compatibles
  repputcs chiptype multiply/divide instruction existence

Example `repputcs 054 use`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repputcs.sh chiptype multiply/divide instruction existence

Example `/bin/sh repputcs.sh 054 use`

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

- For IBM PC/AT and compatibles
  repselo chiptype multiply/divide instruction existence

Example `repselo 054 use`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repselo.sh chiptype multiply/divide instruction existence

Example `/bin/sh repselo.sh 054 use`

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

- For IBM PC/AT and compatibles
  repselon chiptype multiply/divide instruction existence

Example `repselon 054 use`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repselon.sh chiptype multiply/divide instruction existence

Example `/bin/sh repselon.sh 054 use`

(9) Interrupt vector table update

- For IBM PC/AT and compatibles
  repvect chiptype multiply/divide instruction existence

Example `repvect 054 use`

- For HP9000 series 700, SPARCstation Family
  /bin/sh repvect.sh chiptype multiply/divide instruction existence

Example `/bin/sh repvect.sh 054 use`

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 78K0 Series. Not all of these extended functions are always effective. Therefore, it is recommended to use only the effective ones according to the user's purpose. For the effective use of the extended functions, refer to **CHAPTER 13 EFFECTIVE UTILIZATION OF COMPILER** along with this chapter.

C source programs created by using the extended functions of the 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" stands for the 78K0 Series real-time OS.

11.1 Macro Names

This C compiler has two types of macro names: those indicating the series names for target devices and those indicating device names (processor types). These macro names are specified according to the option at compile time to output object code for a specific target device or according to the processor type in the C source. In the example below, **_ _K0_ _** and **_ _054_** are specified.

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

[Example]

Compile option

```
>CC78K0  -C054  prime.c …
```

Specification of device type:

```
#pragma pc (054)
```

11.2 Keywords

This C compiler is added with the following tokens as keywords to realize the extended function. These tokens cannot be used as labels nor variable names as well as ANSI-C keywords. All the keywords must be described in lowercase letters. A keyword containing any uppercase letter is not interpreted as such by the C compiler.

The following shows the list of keywords added to 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 **2.2 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
	noauto	**noauto** functions
_ _leaf	**norec**	**norec**/**_ _leaf** functions
_ _boolean	**boolean**	**boolean** type/**_ _boolean** type variables
	bit	**bit** type variables
_ _interrupt		Hardware interrupt
_ _interrupt_brk		Software interrupt
_ _banked 1 to 15		Bank function
_ _asm		ASM statements
_ _rtos_interrupt		Handler to allocate for RTOS
_ _pascal		Pascal function
_ _flash		Firmware ROM function
_ _flashf		**_ _flashf** function
_ _directmap		Absolute address allocation specification
_ _temp		Temporary variable

(1) Functions

The keywords **callt**, **_ _callt**, **callf**, **_ _callf**, **noauto**, **norec**, **_ _leaf**, **_ _interrupt**, **_ _interrupt_brk**, **_ _rtos_interrupt**, **_ _banked 1 to 15**, **_ _flash**, **_ _flashf**, and **_ _pascal** are attribute qualifiers.

These keywords must be described before any function declaration. The format of each attribute qualifier is shown below.

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

[Example]

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

Attribute qualifier specifications are limited to those listed below. (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, the qualifier added with '_ _' 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`
- `_ _banked 1 to _ _banked 15`
- `_ _rtos_interrupt`
- `_ _pascal`
- `_ _pascal noauto`
- `_ _pascal callt`
- `_ _pascal callf`
- `noauto _ _pascal`
- `callt _ _pascal`
- `callf _ _pascal`
- `callt noauto _ _pascal`
- `callf noauto _ _pascal`
- `_ _flash`
- `_ _flashf`

(2) Variables

- The same regulations apply to the **sreg** or **_ _sreg** specification as to the **register** in C language (refer to **11.5 (3) How to use the saddr area** for details).
- The same regulations apply to the **bit**, **boolean** or **_ _boolean** specification as to the **char** or **int** type specifier in C language.
  However, these types can be specified only to the variables defined outside a function (external variables).
- The same regulations apply to the **_ _directmap** specification as to the type qualifier in C language (refer to **11.5 (45) Absolute address allocation specification** for details).
- The same regulations apply to the **_ _temp** specification as to the type qualifier in C language (refer to **11.5 (47) Temporary variables** for details).

11.3 Memory

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

(1) Memory model

Since memory space is a maximum of 64 KB, the model is 64 KB with code division and data division combined. Code division can have memory space larger than 64 KB by using the bank function.

(2) Register bank

- The register bank is set to ‘RB0’ at start-up (set in the start-up routine of this compiler). The register bank 0 is made always used (unless the register bank is changed) by this setting.
- The specified register bank is set at the start of the interrupt function that has specified the change of the register bank.

(3) Memory space

This C compiler uses memory space as shown below.

Table 11-2 Utilization of Memory Space

(a) Normal model (default)

Address		Use		Size (bytes)
00	40 to 7FH	CALLT table		64
0800 to 0FFFH		CALLF entry		2048
FE	20 to B7H	**sreg** variables, **boolean** type variables		152
FE	B8 to BFH	Arguments of runtime library		8
FE	C0 to C7H	Arguments of **norec** functions		8
FE	C8 to CFH	Automatic variables of **norec** functions		8
FE	D0 to DFH	Register variables		16
FE	E0 to F7H	RB3 to RB1	Work registers[Note 1]	24
	F8 to FFH	RB0	Work registers	8
FF	00 to FFH	**sfr** variables		256

(b) Static model (at -SM16 specification)

Address		Use		Size (bytes)
00	40 to 7FH	CALLT table		64
0800 to 0FFFH		CALLF entry		2048
FE	20 to CFH	**sreg** variables, **boolean** type variables		176
FE	D0 to DFH	Shared area[Note 2]		16
FE	Consecutive areas between 20 and DFH	For arguments, automatic variables, and work[Note 3]		8
FE	E0 to F7H	RB3 to RB1	Work registers[Note 1]	24
	F8 to FFH	RB0	Work registers	8
FF	00 to FFH	**sfr** variables		256

Notes 1. Used when a register bank is specified.

2. The area used by the compiler differs depending on the parameters of the **-SM** option. The area not used as a shared area can be used as **sreg** and **boolean** type variables.

3. Valid only when the static model expansion specification option (**-ZM**) is specified.

11.4 #pragma Directive

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

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 11-3 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 (8) ASM statements**
#pragma vect **#pragma interrupt**	Describes interrupt processing in C → **11.5 (9) Interrupt functions**
#pragma di **#pragma ei**	Describes DI/EI instructions in C → **11.5 (11) Interrupt functions**
#pragma halt **#pragma stop** **#pragma nop** **#pragma brk**	Describes CPU control instructions in C → **11.5 (12) CPU control instruction**
#pragma access	Uses absolute address access functions → **11.5 (14) Absolute address access function**
#pragma section	Changes compiler output section name and specify section location → **11.5 (16) Changing compiler output section name**
#pragma name	Changes module name → **11.5 (18) Module name changing function**
#pragma rot	Uses rotate function → **11.5 (19) Rotate function**
#pragma mul	Uses multiplication function → **11.5 (20) Multiplication function**
#pragma div	Uses division function → **11.5 (21) Division function**
#pragma bcd	Uses BCD operation function → **11.5 (22) BCD operation function**
#pragma tbl	Uses table jump function → **11.5 (24) Table jump function**
#pragma opc	Uses data insertion function → **11.5 (25) Data insertion function**
#pragma rtos_interrupt	Uses interrupt handler for real-time OS (RX78K0) → **11.5 (26) Interrupt handler for real-time OS (RTOS)**
#pragma rtos_task	Uses task function for real-time OS (RX78K0) → **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 (35) Flash area branch table**
#pragma ext_func	Calls a function to the flash area from the boot area → **11.5 (36) Function of function call from boot area to flash area**
#pragma realregister	Uses register direct reference function → **11.5 (40) Register direct reference function**
#pragma hromcall	Uses on-chip firmware self-programming direct subroutine call function → **11.5 (42) On-chip firmware self-programming subroutine direct call function**
#pragma inline	Expands the standard library functions memcpy and memset inline → **11.5 (44) Memory manipulation function**

11.5 How to Use Extended Functions

This section describes each of these extended functions in the following format:

FUNCTION:
Outlines a function that can be implemented with the extended function.

EFFECT:
Explains the effect brought about by the extended function.

USAGE:
Explains how to use the extended function.

EXAMPLE:
Indicates an application example of the extended function.

RESTRICTIONS:
Explains restrictions if any on the use of the extended function.

EXPLANATION:
Explains the above application example.

COMPATIBILITY:
Explains the compatibility of a C source program developed by another C compiler when it is to be compiled with 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 40H to 7FH.
- When the **callt** table is used exceeding the number of callt attribute functions permitted, a compile error will occur.
- The **callt** table is used by specifying the **-QL** option. For that reason, the number of callt attributes permitted per 1 load module and the total in the linking modules is as shown in **Table 11-4**.
- In the case of devices without multiply and divide instructions, two **callt** tables are used for executing multiply and divide, so the maximum number of tables is reduced by two.
- When the option for using the library that supports prologue/epilogue (**-ZD** option) is specified, the **-QL4** option cannot be used. Also, because two callt entries are used by the library that supports prologue/epilogue in the case of a normal model and up to ten in the case of a static model, the maximum number of **callt** entries is reduced two in the case of a normal model and no more than ten in the case of a static model.

Table 11-4 The Number of callt Attribute Functions That Can Be Used When the -QL Option Is Specified

Option	-QL1	-QL2	-QL3	-QL4
Normal model	31	29	9	0
Static model	32	28	19	10

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

Table 11-5 Restrictions on callt Function Usage

callt Function	Restriction Value	
	Normal Model	Static Model
Number per load module	31 max.	32 max.
Total number in linked module	31 max.	32 max.

Caution When normal model is specified, one callt table is used in the bank function call library. For details of bank function, refer to **11.5 (23) Bank function**.

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 compiler)

ca1 module

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

ca2 module

```
       PUBLIC    _tsub            ; Declaration
       PUBLIC    ?tsub            ;
@@CALT CSEG      CALLT0           ; Allocation to segment
?tsub: DW        _tsub
@@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>

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

<From 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 (HL) and **saddr** area **(_@KREG00** to **_@KREG15**). Saves and restores registers or **saddr** area during the preprocessing/ postprocessing of the module that declared a register.
- In the case of the static model, the allocation is performed based on the number of times referenced. Therefore, it is undefined to which register or **saddr** area the register variable is allocated.
- For the details of the allocation of register variables, refer to **11.7 Function Call Interface**.
- Register variables are allocated to different areas depending on the compile condition as shown below (for each option, refer to the **CC78K0 C Compiler Operation User's Manual**).

  1. In the case of the normal model, the register variables are allocated in the declared sequence to register **HL** or the **saddr** area [FED0H to FEDFH]. If there is no stack frame, register variables are allocated to register **HL**. Only when the **-QR** option is specified, register variables are allocated to the **saddr** area.
  2. In the case of the static model, the register variables are allocated to register **DE** or **_@KREGxx** secured by **-SM** specification according to the number of times referenced. Only when the **-ZM2** option is specified, register variables are allocated to the **_@KREGxx**. For details of the **-ZM2** option, refer to **11.5 (46) Static model expansion specifications**.

EFFECT

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

USAGE

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

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

EXAMPLE

```
void main (void) {
        register unsigned char c ;
               .
               .
               .
}
```

Register Variables register

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.

(Normal model)

- **char** uses half the area of other types. **long/float/double/long double** use twice the area. Between **char**s there are byte boundaries but in other cases, there are word boundaries.
- In the case of **int/short** and pointer, a maximum of 8 variables per function is usable. From the 9th variable, the register variables are assigned to the normal memory.
- In the case of a function without a stack frame, a maximum of 8 variables per function is usable for **int/short** and pointer. From the 9th variable, the register variables are assigned to the normal memory.

(Static model)

- **char** uses half the area of other types.
- In the case of **int/short** and pointer, a maximum of 1 variable per function is usable.
- From the 2nd variable, the register variables are assigned to the normal memory.
- The register variables are invalid for **long/float/double/long double**.

Table 11-6 Restrictions on Register Variables Usage

Data Type	Usable Number (per Function)	
	Normal Model	Static Model
int/short	8 variables max.	1 variable max.
Pointer	8 variables max. (9 variables max. if function without stack frame)	1 variable max.

Register Variables register

EXAMPLE

(C source)

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

(Output object of compiler)

- When the **-SM** option is not specified (Example of register variable allocation to register **HL** and the **saddr** area)
  The following labels are declared by the startup routine (Refer to **APPENDIX A LIST OF LABELS FOR saddr AREA**).

```
  EXTRN     _@KREG00          ; References the saddr area to be used
_main:
  push      hl                ; Saves the contents of the register at the beginning of the function
  movw      ax, _@KREG00      ; Saves the contents of the saddr at the beginning of the function
  push      ax                ;

  movw      hl, #00H          ; The following codes are output in the middle of the function
  movw      _@KREG00, #01H    ;
  movw      ax, _@KREG00      ;
  xch       a, x              ;
  add       l, a              ;
  xch       a, x              ;
  addc      h, a              ;
  call      !_func            ;

  pop       ax                ; Restores contents of the saddr at the end of the function
  movw      _@KREG00, ax      ;
  pop       hl                ; Restores contents of the register at the end of the function
  ret
```

Register Variables register

- When the **-SM** option is specified (Example of register variable allocation to register DE)

```
_main:
    push    de                  ; Saves the contents of the register at the beginning of the function

    movw    de, #00H            ;
    movw    ax, #01H            ;
    movw    !?L0003, ax         ;
    xch     a, x                ;
    add     e, a                ;
    xch     a, x                ;
    addc    d, a                ;
    call    !_func              ;
    pop     de                  ; Restores the contents of the register at the end of the function
    ret
```

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>

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

<From this C compiler to another C compiler>

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

(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 (called **sreg** variable) declared with keyword **sreg** or **_ _sreg** are automatically allocated to **saddr** area [FE20H to FEB7H] (normal model) and [FE20H to FECFH] (static model) with relocatability. When those variables exceed the area shown above, a compile 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 **boolean** type variable automatically.
- **sreg** variables declared without an initial value take 0 as the initial value.
- Of the **sreg** variables declared in the assembler source, the **saddr** area [FE20H to FEFFH] can be referred to. The area [FEB8H to FEFFH] (normal model) and [FED0H to FEFFH] (static model) are used by compiler so that care must be taken (refer to **Table 11-2**).

EFFECT

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

USAGE

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

```
sreg type-name  variable-name / sreg static type-name  variable-name
_ _sreg type-name  variable-name / _ _sreg static type-name  variable-name
```

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

```
extern sreg type-name  variable-name / extern _ _sreg type-name  variable-name
```

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.
- **char** type uses a half the space of other types and **long/float/double/long double** types use a space twice as wide as other types.
- Between **char** types there are byte boundaries, but in other cases, there are word boundaries.
- When **-ZA** is specified, only **_ _sreg** is enabled and **sreg** is disabled.
- In the case of **int/short** and pointer, a maximum of 76 variables per load module is usable (when **saddr** area [FE20H to FEB7H] is used). Note that the number of usable variables decreases when **bit** and **boolean** type variables are used (normal model).
- In the case of **int/short** and pointer, a maximum of 88 variables per load module is usable (when **saddr** area [FE20H to FECFH] is used). Note that the number of usable variables decreases when **bit**, **boolean** type variables, and shared areas are used (static model).

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

Table 11-7 Restrictions on sreg Variables Usage

Data Type	Usable Number of **sreg** Variables (per load module)	
	When **saddr** Area [FE20H to FEB7H] is Used	When **saddr** Area [FE20H to FECFH] is Used
int/short, pointer	76 variables max.[Note]	88 variables max.[Note]

Note When **bit** and **boolean** type variables are used, the usable number is decreased.

EXAMPLE

(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 **sreg** variable that the user creates. If **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    SADDRP              ; Allocation to segment
            ORG     0FE20H              ;
_hsmm0:     DS      (2)                 ;
_hsmm1:     DS      (2)                 ;
_hsptr:     DS      (2)                 ;
```

(Output object of compiler)

The following codes are output in the function.

```
movw        ax,_hsmm0
xch         a,x
sub         a,_hsmm1
xch         a,x
subc        a,_hsmm1+1
movw        _hsmm0,ax
```

COMPATIBILITY

<From another C compiler to this C compiler>

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

<From this C compiler to another C compiler>

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

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 **saddr** area regardless of whether **sreg** declaration is made or not.
- Depending on the value of n, the external **static** variables and external **static** variables to allocate can be specified as follows.

Table 11-8 Variables Allocated to saddr Area by -RD Option

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

- Variables declared with the keyword **sreg** are allocated to the **saddr** area, regardless of the above specification.
- The above rule also applies to variables referenced by **extern** declaration, and processing is performed as if these variables were allocated to the **saddr** area.
- The variables allocated to the **saddr** area by this option are treated in the same manner as the **sreg** variable. The functions and restrictions of these variables are as described in (1).

METHOD OF SPECIFICATION

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

RESTRICTIONS

- In **-RD [n]** option, modules specifying different **n** value cannot be linked 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 **saddr** area regardless of with/without **sreg** declaration.
- Depending on the value of n, the internal static variables to allocate can be specified as follows.

Table 11-9 Variables Allocated to saddr Area by -RS Option

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

- Variables declared with the keyword **sreg** are allocated to the **saddr** area regardless of the above specification.
- The variables allocated to the **saddr** area by this option are handled in the same manner as the **sreg** variable. The functions and restrictions for these variables are as described in (1).

METHOD OF SPECIFICATION

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

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

Usage of saddr Area -RK

(4) Usage with saddr automatic allocation option for arguments/automatic variables

FUNCTION

- Arguments and automatic variables (except **const** type) are automatically allocated to the **saddr** area regardless of whether or not a **sreg** declaration exists.
- The arguments and automatic variables to be allocated are specified using the values of n.

Table 11-10 Variables Allocated to saddr Area by -RK Option

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

- Variables declared with **sreg** are allocated to the **saddr** area regardless of the above specifications.
- Variables allocated to the **saddr** area by this option are handled in the same way as **sreg** variables.
- Modules that have different n values specified in the **-RK [n]** option can be linked.

USAGE

- Specify the **-RK [n]** option (where n is 1, 2, or 4).

RESTRICTIONS

- Only the static model is supported. When the **-SM** option is not specified, a warning message is output and the automatic allocation is ignored.
- Arguments/variables that have been declared register variable are not allocated to the **saddr** area.
- When the **-QV** option is specified simultaneously, allocation to register DE has priority.

Usage of saddr Area -RK

EXAMPLE

(C source)

```
sub(int hsmarg)
{
     int hsmauto;
     hsmauto = hsmarg;
}
```

(Output object of compiler)

```
@@DATS DSEG   SADDRP
?L0003:DS     (2)
@@CODE CSEG
_sub:
       movw   ?L0003, ax   ; hsmauto
       ret
```

(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 78K0 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 value (undefined).
- A write check will be performed on read-only **sfr** variables.
- A read check will be performed on write-only **sfr** variables.
- Assignment of an illegal data to an **sfr** variable will result in a compile error.
- The **sfr** names that can be used are those allocated to an area consisting of addresses FF00H to FFFFH.

EFFECT

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

USAGE

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

```
#pragma  sfr
```

- The **#pragma sfr** directive must be described at the beginning of the C source line. If **#pragma PC** (processor type) is specified, however, describe **#pragma sfr** after that.
  The following statement and directives may precede the #pragma sfr directive:
  - Comment statement
  - Preprocessor directives which do not define nor refer to a variable or function
- In the C source program, describe an **sfr** name that the device has as is (without change). In this case, the **sfr** need not be declared.

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 _ _K0_ _
        #pragma sfr
#endif

void main()
{
        P0 -= ADCR;
        /* ADCR = 10;       ==> error */
}
```

(Output object of compiler)

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

```
mov     a, P0
sub     a, ADCR
mov     P0, a
```

Usage of sfr Area sfr

COMPATIBILITY

<From another C compiler to this C compiler>

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

<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 a **sfr** variable. The following shows an example.

```
#ifdef _ _K0_ _
    #pragma sfr
#else
/* Declaration of variables */
unsigned char P0;
#endif

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

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

(5) noauto functions

noauto Functions — noauto

FUNCTION

- **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 or **saddr** area (FEDCH to FEDFH) for register variables. If there is an argument that cannot be allocated to registers, a compile error occurs.
- Automatic variables can be used only if all the automatic variables are allocated to the registers or **saddr** area for register variable-use left over after argument allocation (only when **-ZO** is not specified).
- The **noauto** function allocates arguments to the **saddr** area for register variable-use, but only if the **-QR** option has been specified during the compilation.
- The **noauto** function stores arguments other than arguments allocated to the register in the **saddr** area for register variable-use, and stores the arguments' description in ascending sequence (Refer to **APPENDIX A LIST OF LABELS FOR saddr AREA**).
- The code for calling the **noauto** function output is the same code as the code for calling a normal function (when **-ZO** option is not specified). When calling the noauto function, before storing arguments in the register, save the contents of the register in the **saddr** area for register argument-use, and the contents of the **saddr** area for the register arguments in the stack. Then, after returning from the **noauto** function, output the code that will restore the register with the arguments stored in it and the **saddr** area for register variable-use (when **-ZO** option is specified).
- When the **-SM** option is specified, a warning message is only output to the line in which **noauto** is described first, and all the **noauto** functions are handled as normal functions.

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
```

noauto Functions noauto

RESTRICTIONS

- When the **-ZO** option is specified, automatic variables cannot be used inside noauto function, and so are the register variables.
- When the **-ZA** option is specified, **noauto** function is disabled.
- The arguments and automatic variables of **noauto** function (only when the **-ZO** option is not specified) have restrictions for their types and numbers. The following shows the types of arguments that can be used inside a **noauto** function. Arguments other than **long/signed long/unsigned long**, **float/double/long double** are allocated to register HL.

> - 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**

- When **char** type arguments are allocated to register HL, only **char** type 1 argument can be allocated (only when the **-ZO** option is specified).
- The number of arguments and automatic variables that can be used (when the **-ZO** option has not been specified) is a maximum of 6 bytes in total size.
- These restrictions are checked at the time of compile.
- If arguments are declared with a **register**, the **register** declaration is ignored.

EXAMPLE

(C source)

When the **-QR** option is specified

```
noauto short nfunc(short a, short b, short c);
short l, m;
void main()
{
          static short ii, jj, kk;
          l = nfunc(ii, jj, kk);
}
noauto short nfunc(short a, short b, short c)
{
          m = a + b + c;
          return(m);
}
```

noauto Functions noauto

(Output object of compiler)

```
@@CODE      CSEG
_main:
; line        5:  static short ii, jj, kk;
; line        6:  l = nfunc(ii, jj, kk);
 movw       ax, !?L0005          ; kk
 push       ax
 movw       ax, !?L0004          ; jj
 push       ax
 movw       ax, !?L0003          ; ii
 call       !_nfunc              ; Calls function nfunc(a,b,c)
 pop        ax
 pop        ax
 movw       ax, bc
 movw       !_l, ax              ; Assigns return value to external variable l
; line        7:  }
 ret
; line        8:  noauto short nfunc (short a, short b, short c)
; line        9:  {
_nfunc:
 push       hl                   ; Saves HL
 xch        a, x                 ;
 xch        a, _@KREG12          ; Sets argument a to _@KREG12
 xch        a, x                 ;
 xch        a, _@KREG13          ;
 push       ax                   ; Saves _@KREG12
 movw       ax, _@KREG14         ;
 push       ax                   ; Saves _@KREG14
 movw       ax, sp               ;
 movw       hl, ax               ;
 mov        a,[hl+10]            ;
 xch        a, x                 ;
 mov        a,[hl+11]            ;
 movw       _@KREG14, ax         ; Sets argument c to _@KREG14
 mov        a,[hl+8]             ;
 xch        a, x                 ;
 mov        a,[hl+9]             ;
 movw       hl, ax               ; Sets argument b to HL
```

noauto Functions noauto

(Output object of compiler …continued)

```
; line        10: m = a + b + c;
  movw      ax, hl              ;
  xch       a, x                ;
  add       a, _@KREG12         ;
  xch       a, x                ;
  addc      a, _@KREG13         ;
  xch       a, x                ;
  add       a, _@KREG14         ;
  xch       a, x                ;
  addc      a, _@KREG15         ; Adds b(HL) and c(_@KREG14)to a(_@KREG12)
  movw      !_m, ax             ; Assigns operation result to external variable m
; line        11: return(m)     ;
  movw      bc, ax              ; Returns the contents of external variable m
  pop       ax                  ;
  movw      _@KREG14, ax        ; Restores _@KREG14
  pop       ax                  ;
  movw      _@KREG12, ax        ; Restores _@KREG12
  pop       hl                  ; Restores HL
  ret
```

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>

- The C source program need not be modified if the keyword **noauto** is not used.
- To change variables to **noauto** variables, modify the program according to the procedure described in **USAGE** above.

<From this C compiler to another C compiler>

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

(6) norec functions

norec Functions norec

FUNCTION

- A function that does not call another function by itself can be changed to a **norec** function.
- With **norec** functions, code for preprocessing and post-processing (stack frame formation) is not output.
- The arguments of norec function are allocated to registers and **saddr** area (FEC0H to FEC7H) for arguments of **norec** function.
- If arguments cannot be allocated to registers and **saddr** area, a compile error occurs.
- Arguments are stored either in the register or the **saddr** area (FEC0H to FEC7H) and the **norec** function is called.
- Automatic variables are allocated to the **saddr** area (FEC8H to FECFH) and so are the register variables.
- The **saddr** area is not used for allocation when the **-QR** option is specified during compilation.
- If arguments other than **long/float/double/long double** types are used, the first argument is stored in register AX, the second in register DE, the third and successive arguments are stored in the saddr area. Note that the arguments stored in registers AX and DE are one argument each regardless of the type of argument.
- The argument stored in register AX is copied to register DE if DE does not have the argument stored at the beginning of the norec function. If there is an argument stored in register DE already, the argument stored in AX is copied to **_@RTARG6** and **7**.
- If automatic variables other than **long/float/double/long** double types are used, the arguments that are left after allocation are stored in the declared order; **DE**, **_@RTARG6** and **7**, and **_@NRARG0**, **1**...
  If automatic variables **long/float/double/long double** types are used, the arguments that are left after allocation are stored in the declared order; **_@NRARG0**, **1**...
  The rest of the arguments are stored in the **saddr** area in the declared order (Refer to **APPENDIX A LIST OF LABELS FOR saddr AREA**).

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 Functions 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.
- When the **-SM** option is specified, a warning message is only output to the line in which **norec** is described first, and all the **norec** functions are handled as normal functions.
- The restrictions for arguments and automatic variables are checked at the time of compile, 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.
  norec functions are allocated to the **saddr** area consecutively if between **char/signed char/unsigned char**, however if connected to other types, allocation is performed in two-byte alignment.

- 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`**

(When the **-QR** option is not specified)

- The number of arguments that can be used in a **norec** function is 2 variables, if other than **long/float/double/long double** types. Arguments cannot be used for **long/float/double/long double** types.
- Automatic variables can use the area that is the combined total of the number of bytes remaining unused by arguments. If types other than **long/float/double/long double** are used, automatic variables can use up to 4 bytes. Arguments can not be used for **long/float/double/long double** types.

(When the **-QR** option is specified)

- The number of arguments is 6 variables, if types other than **long/float/double/long double** are used, and 2 variables if **long/float/double/long double** types are used.
- Automatic variables can use the area that is the combined total of the number of bytes remaining unused by arguments and the number of **saddr** area bytes. If types other than **long/float/double/long double** are used, automatic variables can use up to 20 bytes and if **long/float/double/long double** types are used, automatic variables can use up to 16 bytes.
- These restrictions are checked at the time of compilation and an error will occur if not satisfied.

norec Functions norec

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 Functions norec

(Output object of compiler)

When the **-QR** option is specified

```
        EXTRN       _@NRARG0            ; References saddr area to be used
        EXTRN       _@NRARG1            ;
        EXTRN       _@NRARG6            ;
            .
            .
            .
        _@NRARG0    ← m                 ; Stores argument to saddr area
            .
            .
            .
        de          ← 1                 ; Stores argument to DE
            .
            .
            .
        ax          ← k                 ; Stores argument to AX
        call        !_rout              ; Calls norec function

   _rout:
        movw        _@RTARG6, ax
                                        ; Receives argument from saddr area
        mov         c, #02H
        xch         a, x
        add         a, a
        xch         a, x
        rolc        a, 1
        dbnz        c, $$-5
        xch         a, x
        add         a, _@NRARG1         ; Use automatic variables of saddr area
        xch         a, x                ;
        addc        a, _@NRARG1+1       ; Use automatic variables of saddr area
        movw        bc, ax              ;
        ret
```

norec Functions norec

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>

- The C source program need not be modified if the keyword **norec** is not used.
- To change variables to **norec** variables, modify the program according to the procedure described in **USAGE** 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	**bit**
boolean Type Variables	**boolean**
	_ _boolean

FUNCTION

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

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

EFFECT

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

USAGE

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

```
bit variable-name
boolean variable-name
_ _boolean variable-name
```

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

```
extern   bit variable-name
extern   boolean variable-name
extern   _ _boolean variable-name
```

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

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

bit Type Variables — bit
boolean Type Variables — boolean, _ _boolean

RESTRICTIONS

- An operation on two **bit** or **boolean** type variables is performed by using the CY (Carry) flag.
  For this reason, the contents of the carry flag between statements are not guaranteed.
- Arrays cannot be defined or referenced.
- A **bit** or **boolean** type variable cannot be used as a member of a structure or union.
- This type of variable cannot be used as the argument type of a function.
- A **bit** type variable cannot be used as a type of automatic variable (other than static model).
- With **bit** type variables only, up to 1216 variables can be used per load module (when **saddr** area [FE20H to FEB7H] is used) (normal model).
- With **bit** type variables only, up to 1408 variables can be used per load module (when **saddr** area [FE20H to FECFH] is used) (static model).
- The variable cannot be declared with an initial value.
- If the variable is described along with **const** declaration, the **const** declaration is ignored.
- Only operations using 0 and 1 can be performed by the operators and constants shown in the following table:
- *, & (pointer reference, address reference), and **sizeof** operations cannot be performed.
- When the **-ZA** option is specified, only **_ _boolean** is enabled.

Table 11-11 Operators Using Only Constants 0 or 1 (with Bit Type Variable)

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

Remark In the case that **sreg** variables are used or if **-RD**, **-RS**, and **-RK** (**saddr** automatic allocation option) options are specified, the number of usable bit type variables is decreased.

bit Type Variables — bit
boolean Type Variables — boolean, _ _boolean

EXAMPLE

(C source)

```
#define ON 1
#define OFF 0

extern bit data1;
extern bit data2;

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

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

(Assembler source)

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

```
PUBLIC    _data1                    ; Declaration
PUBLIC    _data2

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

bit Type Variables / boolean Type Variables — bit, boolean, _ _boolean

(Output object of compiler)

The following codes are output in a function

```
set1    _data1              ; Initialized
clr1    _data2              ; Initialized
bf      _data1, $?L0001     ; Judgment
mov1    CY, _data2          ; Assignment
mov1    _data1, CY          ; Assignment
bf      _data1, $?L0005     ; Logical AND expression
bf      _data2, $?L0005     ; Logical AND expression
```

COMPATIBILITY

<From another C compiler to this C compiler>

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

<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, the **bit** or **boolean** type variables are handled as ordinary variables.).

(8) 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 preprocessor directives **#asm** and **#endasm**.
- **#asm** and **#endasm** lines will not be output.

(b) _ _asm

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

EFFECT

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

USAGE

(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
```

(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 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.

ASM Statements — #asm, #endasm, _ _asm

RESTRICTIONS

- Nesting of **#asm** directives is not allowed.
- If **ASM** statements are used, no object module file will be created. Instead, an assembler source file will be created.
- Only lowercase letters can be described for **_ _asm**. If **_ _asm** is described with uppercase and lowercase characters mixed, it is regarded as a user function.
- When the **-ZA** option is specified, only **_ _asm** is enabled.
- **#asm** - **#endasm** and **_ _asm** block can only be described inside a function of the C source. Therefore, the assembler source is output to **CSEG** of segment name **@@CODE**.

EXAMPLE

(a) #asm - #endasm

(C source)

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

(Output object of compiler)

The assembler source written by the user is output to the assembler source file.

```
@@CODE   CSEG
_main:
         callt  [init]
         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 the C compiler which supports **#asm**, modify the program according to the format specified by the C compiler.
- If the target device is different, modify the assembler source part of the program.

(9) 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)	**saddr** area for arguments/**auto** variables of **norec** function (regardless of whether the arguments or variables are used)
(4)	**saddr** area for run time library (normal model only)

Note, however, that depending on the specification or status of the interrupt function, saving/restoring is performed differently, as follows:

- If no change is specified, codes that change the register bank or saves/restores register contents, and that saves/restores the contents of the **saddr** area are not output regardless of whether to use the codes or not.
- If a register bank is specified, a code to select the specified register bank is output at the beginning of the interrupt function, therefore, the contents of the registers are not saved or restored.
- If no change is not specified and if a function is called in the interrupt function, however, the entire register area is saved or restored, regardless of whether use of registers is specified or not.

(In the case of the normal model)

- If the **-QR** option is not specified at compile time, the **saddr** area for register variable and the **saddr** area for the arguments/**auto** variable of the **norec** function is not used; therefore, the saving/restoring code is not output.
  If the size of the saving code is smaller than that of the restoring code, the restoring code is output.
- **Table 11-12** summarizes the above and shows the saving/restoring area.

Interrupt Functions #pragma vect #pragma interrupt

Table 11-12 Saving/Restoring 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	×	√	×	√	×	×	×	×	×
saddr area for runtime library used	×	×	×	×	×	√	√	√	√
saddr area for all runtime libraries	×	√	√	√	√	×	×	×	×
saddr area for register variable used	×	×	×	√	√	×	×	√	√
All **saddr** area for arguments/**auto** variables of norec function	×	×	×	√	√	×	×	×	×

Stack: Use of stack is specified √: Saved
RBn: Register bank is specified ×: Not saved

(Static model)

- Since the **saddr** area for register variables, the **saddr** area for automatic variables or **norec** function arguments, and the **saddr** area for the runtime library are not used when the **-SM** option is specified during compilation, the save and restore code area is as follows.

Table 11-13 Saving/Restoring Area When Interrupt Function Is Used (Static Model)

Save/Restore Area	NO BANK	With Function Call		Function Not Called	
		Stack	RBn	Stack	RBn
Register used	×	×	×	√	×
All registers	×	√	×	×	×

Stack: Use of stack is specified √: Saved
RBn: Register bank is specified ×: Not saved

- However, when **leafwork 1** to **16** has been specified, the code for saving and restoring the byte number to the stack is output from the higher-level address of shared area at the beginning and end of the interrupt function (Refer to **11.5 (29) Static model** when the **-ZM** option is not specified, and **11.5 (46) Static model expansion specification** when the **-ZM** option is specified).

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.

Interrupt Functions — #pragma vect / #pragma interrupt

EFFECT

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

USAGE

- Specify an interrupt request name, a function name, stack switching, registers, and whether the **saddr** area is saved/restored, with the **#pragma** directive. Describe the **#pragma** directive at the beginning of the C source. The **#pragma** directive is described at the start of the C source (for the interrupt request names, refer to the user's manual of the target device used). For the software interrupt BRK, describe BRK_I.
- To describe **#pragma PC** (processor type), describe this **#pragma** directive after that. The following items can be described before this **#pragma** directive:

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

  (In the case of the normal model)

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

        [stack change specification] Δ [{ stack use specification        }]
                                        [{ No change specification        }]
                                        [{ Register bank specification    }]
```

  (In the case of the static model)

```
#pragmaΔvect (or interrupt)ΔInterrupt request nameΔFunction nameΔ

[{ Shared area save/restore specification }] Δ [{ Stack usage specification     }]
[{ Save/restore target                    }]   [{ No change specification       }]
                                               [{ Register bank specification   }]
```

Interrupt Functions — #pragma vect / #pragma interrupt

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
Shared area save/restore specification	: **leafwork 1** to **16**
Save/restore target	: SAVE_R Save/restore target limited to registers SAVE_RN Save/restore target limited to registers and **_@NRATxx** (when **-SM**, **-ZM** option specified)
Δ	: Space

Caution Since the CC78K0 startup routine is initialized to register bank 0, be sure to specify register banks 1 to 3.
When saving shared area by the leafwork specification, the number of bytes specified needs to be same as the maximum bytes of the shared area secured in the -SM option of all modules.

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.
- The contents of a register may be changed if the following three conditions are satisfied, but the compiler cannot check this. If it is specified to change the register bank, set the register banks so that they do not overlap. If register banks overlap, control their interrupts so that they do not overlap. When NOBANK (no change specification) is specified, the registers are not saved. Therefore, control the registers so that their contents are not lost.
  (1) If two or more interrupts occur
  (2) If two or more interrupts that use the same BANK are included in the interrupt that has occurred
  (3) If NOBANK or a register bank is specified in the description #pragma interrupt ~.
- As the interrupt function, **callt/callf/noauto/norec/_ _callt/_ _callf/_ _leaf/_ _banked 1** to **15/_ _rtos_interrupt/ _ _pascal/_ _flash/_ _flashf** cannot be specified.
- An interrupt function is specified with **void** type (example: **void** func (**void**);) because it cannot have an argument nor return value.
- Even if an **ASM** statement exists in the interrupt function, codes saving all the registers and variable areas are not output. If an area reserved for the compiler is used in the **ASM** statement in the interrupt function, therefore, or if a function is called in the **ASM** statement, the user must save the registers and variable areas.

Interrupt Functions #pragma vect #pragma interrupt

- If a function specifying no change, register bank, or stack change as the saving destination in **#pragma vect/#pragma interrupt** specification is not defined in the same module, a warning message is output and the stack change is ignored. In this case, the default stack is used.
- When stack change is specified, the stack pointer is changed to the location where offset is added to the array name symbol. The area of the array name is not secured by the **#pragma** directive. It needs to be defined separately as global **unsigned 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 keywords **sreg/_ _sreg** are added to the array for stack change, it is regarded that two or more variables with the different attributes and the same name are defined, and a compile error occurs. It is possible to allocate an array in saddr area by the **-RD** option, but code and speed efficiency will not be improved because the array is used as a stack. It is recommended to use the **saddr** area for purposes other than a stack.
- The stack change cannot be specified simultaneously with the no change. If specified so, an error 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.
- If **leafwork 1** to **16** is specified when the **-SM** option is not specified, a warning is output and the save/restore specification of the shared area is ignored.

EXAMPLE

(C source 1)
When register bank is specified

```
#pragma interrupt NMI inter rbl
void inter()
{
    /* Interrupt processing to NMI pin input */
}
```

(Output object of compiler)

```
@@CODE      CSEG
_inter:

    Switching code for the register bank
    Save code of the saddr area for use by the compiler
    Interrupt processing to NMI pin input (function body)
    Restore code of the saddr area used by the compiler
    reti
@@VECT02    CSEG        AT      02H;NMI
```

Interrupt Functions

#pragma vect #pragma interrupt

```
_@vect02:
    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();
    }

  @@CODE          CSEG
  _inter:
            sel   RB2               ; Changes register bank
            push ax                 ; Changes stack pointer
            movw ax, sp             ; Changes stack pointer
            movw sp, #_buff+10      ; Changes stack pointer
            push ax                 ; Changes stack pointer
            movw ax, _@RTARG0       ; Saves saddr used by the compiler
            push ax                 ; Saves saddr used by the compiler
            movw ax, _@RTARG2       ; Saves saddr used by the compiler
            push ax                 ; Saves saddr used by the compiler
            movw ax, _@RTARG4       ; Saves saddr used by the compiler
            push ax                 ; Saves saddr used by the compiler
            movw ax, _@RTARG6       ; Saves saddr used by the compiler
            push ax                 ; Saves saddr used by the compiler
            call !_func
            pop  ax                 ; Restores saddr used by the compiler
            movw _@RTARG6           ; Restores saddr used by the compiler
            pop  ax                 ; Restores saddr used by the compiler
            movw _@RTARG4           ; Restores saddr used by the compiler
            pop  ax                 ; Restores saddr used by the compiler
            movw _@RTARG2           ; Restores saddr used by the compiler
            pop  ax                 ; Restores saddr used by the compiler
```

Interrupt Functions #pragma vect #pragma interrupt

```
          movw  _@RTARG0          ; Restores saddr used by the compiler
          pop   ax                ; Returns the stack pointer to its original position
          movw  sp, ax            ; Returns the stack pointer to its original position
          pop   ax                ; Returns the stack pointer to its original position
          reti

@@VECT06        CSEG  AT      0006H
_@vect06:
          DW     _inter
```

(C source 3)

When a shared area save/restore is specified (static model only)

```
   #pragma interrupt INTP0 inter leafwork4
   void func();
   void inter()
   {
          func();
   }

     EXTRN    _@KREG12
     EXTRN    _@KREG14

@@CODE        CSEG
_inter:
     push     ax                ; Saves register
     push     bc                ; Saves register
     push     hl                ; Saves register
     movw     ax, _@KREG12      ; Saves shared area
     push     ax                ; Saves shared area
     movw     ax, _@KREG14      ; Saves shared area
     push     ax                ; Saves shared area
     call     !_func
     pop      ax                ; Restores shared area
     movw     _@KREG14, ax      ; Restores shared area
     pop      ax                ; Restores shared area
     movw     _@KREG12, ax      ; Restores shared area
     pop      hl                ; Restores register
     pop      bc                ; Restores register
```

Interrupt Functions

#pragma vect
#pragma interrupt

```
          pop     ax              ; Restores register
          reti
     @@VECT06     CSEG  AT        0006H
     _@vect06:
          DW      _inter
```

COMPATIBILITY

<From another C compiler to this C compiler>

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

<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.

(10) 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 function.
- By declaring a function with the **_ _interrupt_brk** qualifier, the function is regarded as a software interrupt function, and execution is returned by the return instruction RETB for software interrupt function.
- A function declared with this qualifier is regarded as (non-maskable/maskable/software) interrupt function, and saves or restores the registers and variable areas (1) and (4) 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)
(4) **saddr** area for run time library

Remark If the **-QR** option is not specified (default) at compile time, save/restore codes are not output because areas (2) and (3) are not used. If the **-SM** option is specified at compilation, save/restore codes are not output because areas (2), (3) and (4) are not used.

EFFECT

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

USAGE

- Describe either **_ _interrupt** or **_ _interrupt_brk** as the qualifier of an interrupt function.

(For non-maskable/maskable interrupt function)
```
_ _interrupt void func() {processing}
```

(For software interrupt function)
```
_ _interrupt_brk void func() {processing}
```

RESTRICTIONS

- The interrupt function cannot specify **callt/callf/noauto/norec/_ _callt/_ _callf/_ _leaf/_ _banked 1** to **15/ _ _rtos_interrupt/_ _pascal/_ _flash/_ _flashf.**

Interrupt Function Qualifier __interrupt __interrupt_brk

CAUTIONS

- The vector address is not set by merely declaring this qualifier. The vector address must be separately set by using the **#pragma vect/interrupt** directive or assembler description.
- The **saddr** area and registers are saved to the stack.
- Even if the vector address is set or the saving destination is changed by **#pragma vect** (or **interrupt**) ..., the change in the saving destination is ignored if there is no function definition in the same file, and the default stack is assumed.
- To define an interrupt function in the same file as the **#pragma vect** (or **interrupt**) ... specification, the function name specified by **#pragma vect** (or **interrupt**) ... is judged as the interrupt function, even if this qualifier is not described (for details of **#pragma vect/interrupt**, refer to **USAGE** of **11.5 (9) Interrupt functions**).

EXAMPLE

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

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

_ _interrupt void inter( );                          /*prototype declaration*/
_ _interrupt_brk void inter_b( );                    /*prototype declaration*/
_ _interrupt void inter( )  {processing};            /*function body*/
_ _interrupt_brk void inter_b( )  {processing};      /*function body*/
```

COMPATIBILITY

<From another C compiler to this C compiler>

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

<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.

(11) Interrupt functions

Interrupt Functions — #pragma DI / #pragma EI

FUNCTIONS

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

EFFECT

- A function disabling interrupts can be created.

USAGE

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

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

```
#ifdef _ _K0_ _
       #pragma DI
       #pragma EI
#endif
```

(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** and **EI** after and before preprocessing/post-processing>

(C source 2)

```
#pragma  DI
#pragma  EI
void main (  )
{
        {
                DI (  );
                function body
                EI (  );
        }
}
```

(Output object of compiler)

```
_main:
        preprocessing
        di
        function body
        ei
        post-processing
        ret
```

COMPATIBILITY

<From another C compiler to this C compiler>

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

<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** and **DI** and **EI** can be used as ordinary function names (example: **#ifdef_ _K0_ _ ... #endif**).
- When an ordinary function is to be used as an interrupt function, modify the program according to the specifications of each compiler.

(12) CPU control instruction

CPU Control Instruction #pragma HALT/STOP/BRK/NOP

FUNCTION

- The following codes are output to the object to create an object file:

(1) Instruction for HALT operation (HALT)
(2) Instruction for STOP operation (STOP)
(3) BRK instruction
(4) NOP instruction

EFFECT

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

USAGE

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

```
(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.

CPU Control Instruction #pragma HALT/STOP/BRK/NOP

EXAMPLE

(C source)

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

(Output object of compiler)

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

COMPATIBILITY

<From another C compiler to this C compiler>

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

<From this C compiler to another C compiler>

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

(13) 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 the ordinary **call** instruction.
- If a function stored in the **callf** area is to be referenced without prototype declaration, the function must be called by the ordinary **call** instruction.
- The callee (the function to be called) is the same as ordinary functions.

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.
- When the **-ZA** option is specified, the **callf** function cannot be defined. When it is defined, an error will occur.

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)

```
<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>

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

<From this C compiler to another C compiler>

- **#define** must be used to allow **callf** functions to be handled as ordinary functions.

(14) Absolute address access function

Absolute Address Access Function #pragma access

FUNCTION

- A code to access the ordinary RAM space is output to the object through direct in-line expansion, not by function call, and an object file can be 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 through 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 function call.
- The following items can be described before **#pragma access**:
  - Comment statement
  - Other **#pragma** directives
  - Preprocessor directive which does neither define nor refer to a variable or function
- The keywords following **#pragma** can be described in either uppercase or lowercase letters.

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;
```

Returns 1-byte contents of address **addr**.

(b)
```
unsigned int peekw (addr);
unsigned int addr;
```

Returns 2-byte contents of address **addr**.

(c)
```
void pokeb (addr, data);
unsigned int addr;
unsigned char data;
```

Writes 1-byte contents of **data** to the position indicated by address **addr**.

(d)
```
void pokew (addr, data);
unsigned int addr;
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, 7);
       pokew (0xfe68, 7);
}
```

(Output assembler source)

```
                    .          .
                    .          .
                    .          .
               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
               mov     !01234H, a
               mov     0FE23H, #05H
               movw    ax, #07H
               movw    !01256H, ax
               movw    0FE68H, #07H
```

Absolute Address Access Function #pragma access

COMPATIBILITY

<From another C compiler to this C compiler>

- The source program need not be modified if a function for absolute address accessing is not used.
- Modify the program according to the procedure described in **USAGE** above if a function for absolute address accessing is used.

<From this compiler to another C compiler>

- Delimit the "**#pragma access**" statement by means of deletion or with **#ifdef**. As a function name, the function name of absolute address accessing can be used.
- To use a function for absolute address accessing, the program must be modified according to the specifications of each compiler (**#asm**, **#endasm**, **asm**, etc.).

(15) 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 specifier, **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>

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

<From this C compiler to another C compiler>

- The source program need not be modified if **unsigned char** is not used as a type specifier.
- Change **unsigned char**, if it is used as a type specifier, into **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

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

EXAMPLE 1

(Bit field declaration)

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

Bit Field Declaration — Bit field declaration

EXPLANATION

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

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)

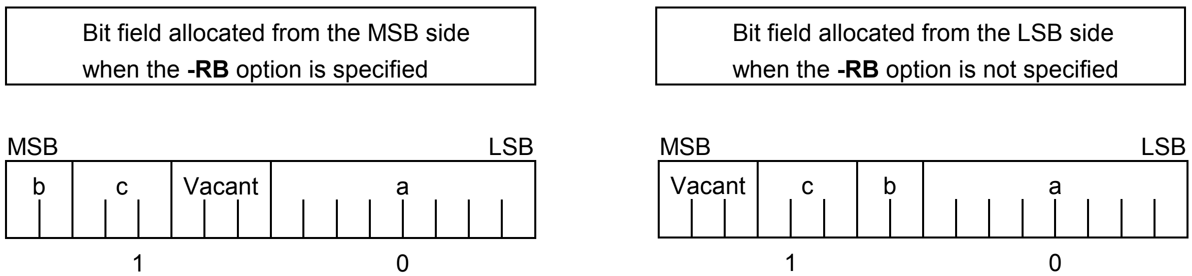
```
struct t {
    char            a;
    unsigned char   b:2;
    unsigned char   c:3;
    unsigned char   d:4;
    int             e;
    unsigned char   f:5;
    unsigned char   g:6;
    unsigned char   h:2;
    unsigned int    i:2;
};
```

Bit Field Declaration

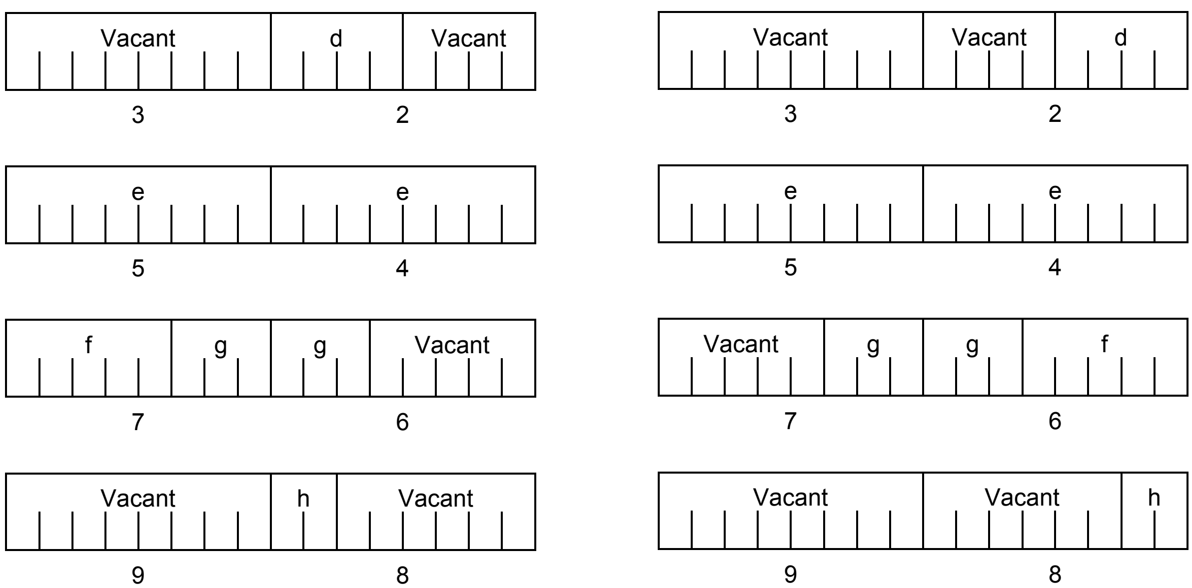
Bit field declaration

EXPLANATION

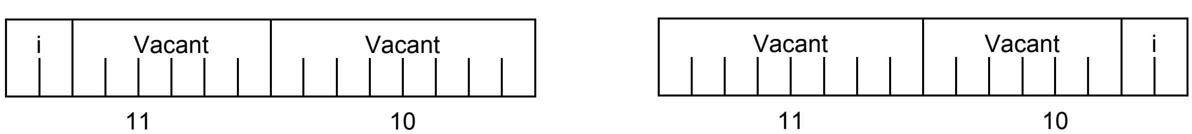
Figure 11-2 Bit Allocation by Bit Field Declaration (Example 2)



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



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



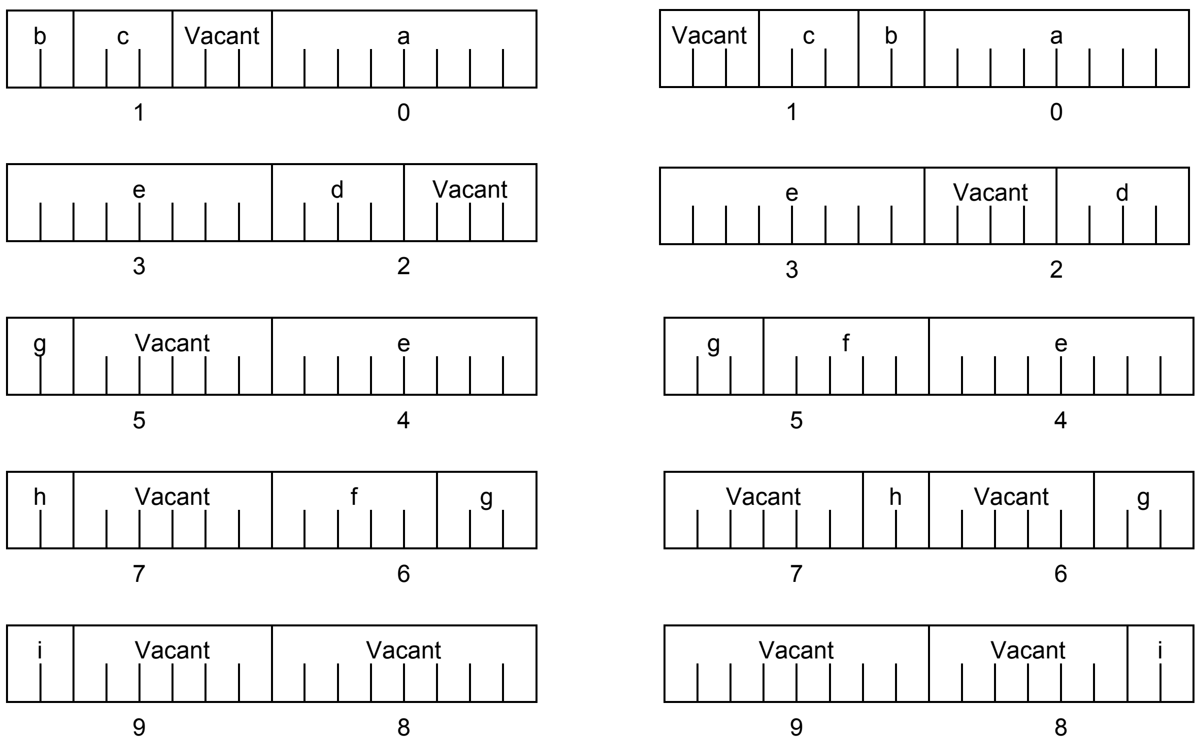
Since i is a bit field of type **unsigned int**, it is allocated in the next word unit.

Bit Field Declaration

Bit field declaration

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

Figure 11-3 Bit Allocation by Bit Field Declaration (Example 2)(with -RC Option Specified)



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

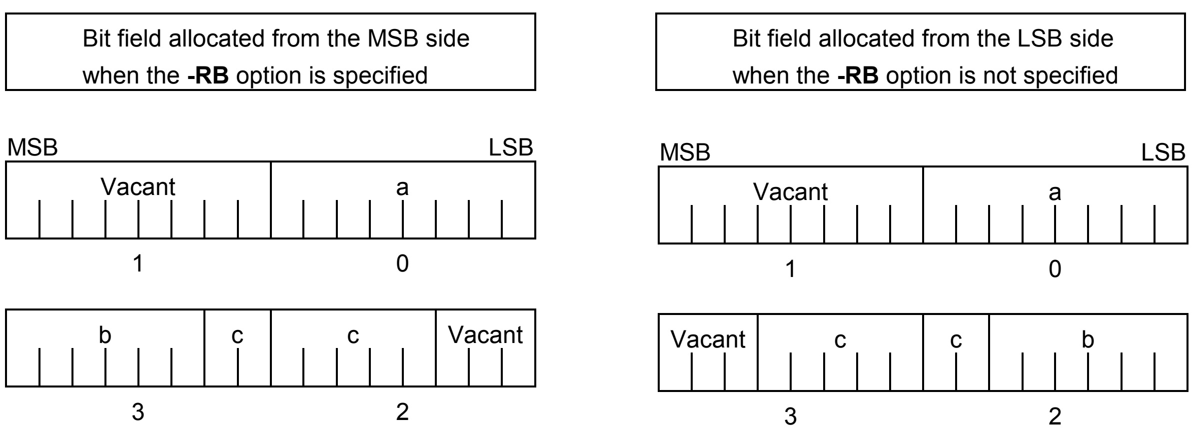
Bit Field Declaration — Bit field declaration

EXAMPLE 3

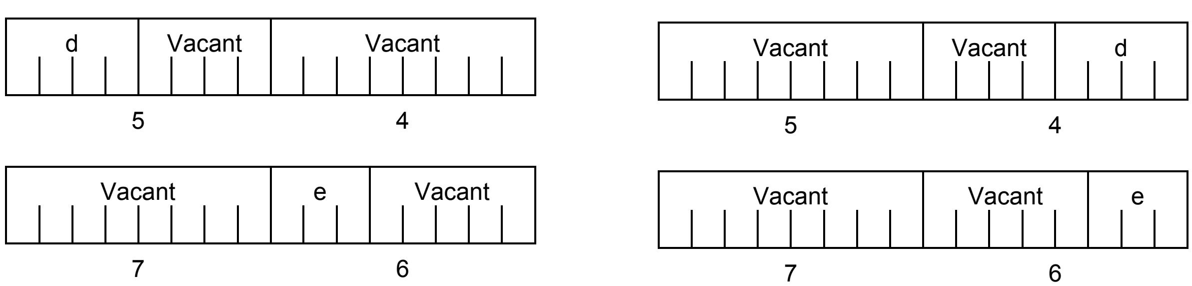
(Bit field declaration)

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

Figure 11-4 Bit Allocation by Bit Field Declaration (Example 3)

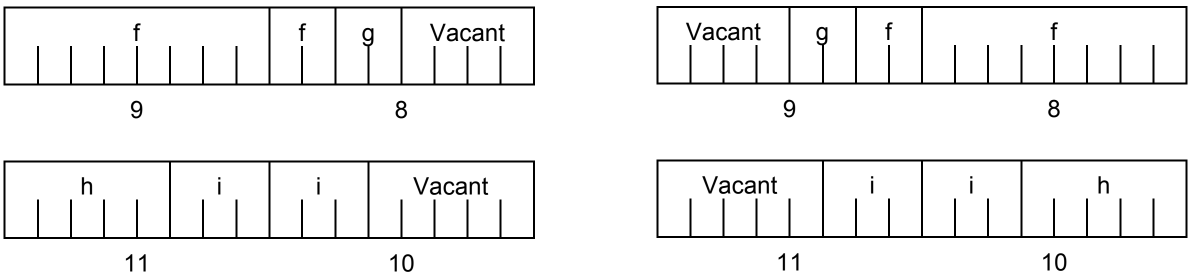


Since b and c are bit fields of type **unsigned int**, they are allocated from the next word unit.
Since d is also a bit field of type **unsigned int**, it is allocated from the next word unit.



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

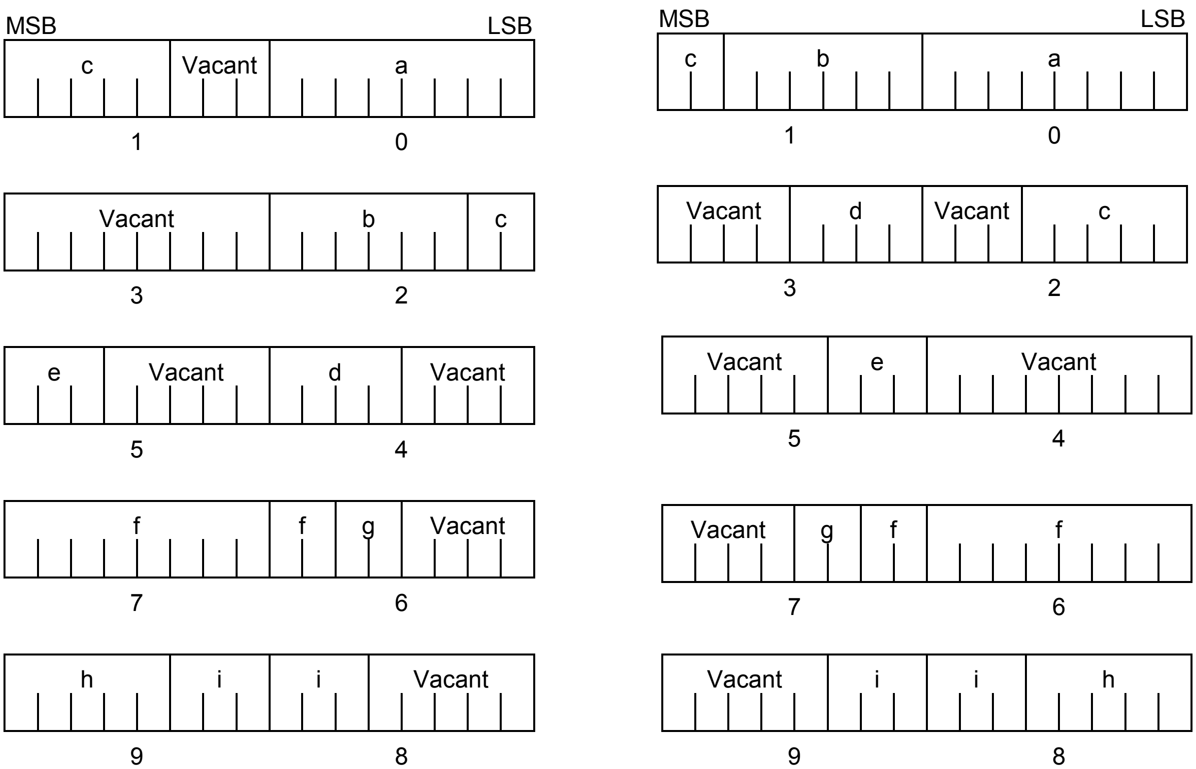
Bit Field Declaration Bit field declaration



f and g, and h and i are each allocated to separate word units.

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

Figure 11-5 Bit Allocation by Bit Field Declaration (Example 3)(with -RC Option Specified)



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>

- The source program need not be modified.

<From this C compiler to another C compiler>

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

(16) 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 **RA78K0 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 are described after the **#pragma** instruction is described, those data are located in the data change section. Another change instruction is possible, and if data are described after the rechange instruction, those data are located in the rechange section. If data defined before a change are redefined after the change, they are located in the rechanged section. Furthermore, this is valid in the same way for **static** variables (within the function).

EFFECT

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

USAGE

- Specify the name of the section which is to be changed, a new section name, and the start address of the section, by using the **#pragma** directive as indicated below.
  Describe this **#pragma** directive at the beginning of the C source.
  Describe this **#pragma** directive after **#pragma PC** (processor type).
  The following items can be described before this **#pragma** directive:

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

However, all sections in **BSEG** and **DSEG**, and the **@@CNST** section in **CSEG** can be described anywhere in the C source, and rechange instructions can be performed repeatedly. To return to the original section name, describe the compiler output section name in the changed section.

Declare as follows at the beginning of the file:

```
#pragma  section  compiler output section name  new section name  [AT start address]
```

- Of the keywords to be described after **#pragma**, be sure to describe the compiler output section name in uppercase letters. **section, AT** can be described in either uppercase or lowercase letters, or in combination of those.

#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 number, specify zero before F. It is therefore 0FFH.

- For sections other than the **@@CNST** section in **CSEG**, that is, sections which locate functions, this **#pragma** instruction cannot be described in other than the beginning of the C source (after the C text is described). If described, it causes an error.
- If this **#pragma** instruction is executed after the C text is described, an assembler source file is created without an object module file being created.
- If this **#pragma** instruction is after the C text is described, a file which contains this **#pragma** instruction and which does not have the C text (including external reference declarations for variables and functions) cannot be included. This results in an error (refer to the error description in Example 1).
- **#include** statement cannot be described in a file which executes this **#pragma** instruction following the C text description. If described, it causes an error. (refer to the following error description in Example 2).
- If **#include** statement follows the C text, this **#pragma** instruction cannot be described after this description. If described, it causes an error.(refer to the following error description in 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
           Post-processing
           ret
```

EXAMPLE 2

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

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

#pragma section… #pragma section…

(EXAMPLE 2 …continued)

```
#pragma section @@DATA @@DATA
// ??DATA2 is closed and processing returns to the default @@DATA
#pragma section @@INIT ??INIT
#pragma section @@R_INIT ??R_INIT
// ROMization is invalidated unless both names (@@INIT and @@R_INIT) are changed.
// This is the user's responsibility.
 int a4;                                  // ??DATA
 sreg int b4;                             // ??DATS
 int c4 = 1;                              // ??INIT and ??R_INIT
 const int d4 = 1;                        // @@CNST
#pragma section @@INIT @@INIT
#pragma section @@R_INIT @@R_INIT
// ??INIT and ??R_INIT are closed and processing returns to the default setting
#pragma section @@BITS ??BITS
 _ _boolean e4;                           // ??BITS
#pragma section @@CNST ??CNST
 char*const p = "Hello";                  // p and "Hello" are both ??CNST
```

EXAMPLE 3

```
#pragma section @@INIT ??INIT1
#pragma section @@R_INIT ??R_INIT1
#pragma section @@DATA ??DATA1
  char c1;
  int i2;
#pragma section @@INIT ??INIT2
#pragma section @@R_INIT ??R_INIT2
#pragma section @@DATA ??DATA2
  char c1;
  int i2 = 1;
#pragma section @@DATA ??DATA3
#pragma section @@INIT ??INIT3
#pragma section @@R_INIT ??R_INIT3
  extern char c1;                  // ??DATA3
  int i2;                          // ??INIT3 and ??R_INIT3
#pragma section @@DATA ??DATA4
#pragma section @@INIT ??INIT4
#pragma section @@R_INIT ??R_INIT4
```

#pragma section… #pragma section…

Restrictions when this **#pragma** directive has been specified after the main C code are explained in the following coding error examples.

CODING ERROR EXAMPLE 1

```
a1.h
 #pragma section @@DATA ??DATA1   // File containing only the #pragma section

a2.h
 extern int func1 (void);
 #pragma section @@DATA ??DATA2   // File containing the main C code followed by the #pragma
                                  // directive

a3.h
 #pragma section @@DATA ??DATA3   // File containing only the #pragma section.

a4.h
 #pragma section @@DATA ??DATA3
 extern int func2 (void);         // File that includes the main C code.

a.c
    #include "a1.h"
    #include "a2.h"
    #include "a3.h"               // ← Results in an error.
                                  // Because the a2.h file contains the main C code followed by this
                                  // #pragma directive, file a3.h, which includes only this #pragma
                                  // directive, cannot be included.
    #include "a4.h"
```

#pragma section… #pragma section…

CODING ERROR EXAMPLE 2

```
b1.h
 const int i;

b2.h
 const int j;
 #include "b1.h"                    // This does not result in an error since it is not file (b.c) in which
                                    // the main C code is followed by this #pragma directive.
b.c
 const int k;
 #pragma section @@DATA ??DATA1
 #include "b2.h"                    // ←  Results in an error
                                    // Since an #include statement cannot be coded afterward in file
                                    // (b.c) in which the main C code is followed by this #pragma
                                    // directive.
```

CODING ERROR EXAMPLE 3

```
c1.h
 extern int j;
 #pragma section @@DATA ??DATA1     // This does not result in an error since the #pragma directive is
                                    // included and processed before the processing of c3.h.
c2.h
 extern int k;
 #pragma section @@DATA ??DATA2     // ← Results in an error.
                                    // This #include statement is specified after the main C code in
                                    // c3.h, and the #pragma directive cannot be specified afterward.
c3.h
 #include "c1.h"
 extern int i;
 #include "c2.h"
 #pragma section @@DATA ??DATA3     // ← Results in an error.
                                    // This #include statement is specified after the main C code, and
                                    // the #pragma directive cannot be specified afterward.
c.c
 #include "c3.h"
 #pragma section @@DATA??DATA4      // ← Results in an error.
                                    // This #include statement is specified after the main C code in
                                    // c3.h, and the #pragma directive cannot be specified afterward.
 int i;
```

#pragma section… #pragma section…

COMPATIBILITY

<From another C compiler to this C compiler>

- The source program need not be modified if the section name change function is not supported.
- To change the section name, modify the source program according to the procedure described in **USAGE** above.

<From this C compiler to another C compiler>

- Delete or delimit **#pragma section** ... with **#ifdef**.
- To change the section name, modify the program according to the specifications of each compiler.

RESTRICTIONS

- A section name that indicates a segment for vector table (e.g., **@@VECT02**, etc.) must not be changed.
- If two or more sections with the same name as the one specifying the **AT** start address exist in another file, a link error occurs.
- Section names (**@@BANK1**, etc.) that indicate segments for bank function use cannot be changed.
- When changing compiler output section names **@@DATS**, **@@BITS**, and **@@INIS**, limit the range of the specified address within 0FE20H to 0FEB7H.

CAUTION

- A section is equivalent to a segment of the assembler.
- The compiler does not check whether the new section name is in duplicate with another symbol. Therefore, the user must check to see whether the section name is not in duplicate by assembling the output assemble list.
- If a section name (*) related to ROMization is changed by using **#pragma section**, the start-up routine must be changed by the user on his/her own responsibility.
- When the **-ZF** option has been specified, each section name is changed so that the second ‘@’ is replaced with ‘E’.

(*) ROMization-related section name

```
@@R_INIT, @@R_INIS, @@INIT, @@INIS
```

The start-up routine to be used when a section related to ROMization is changed, and an example of changing the end module are described later.

#pragma section… #pragma section…

[Examples of Changing Start-up Routine in Connection with Changing Section Name Related to ROMization]

Here are examples of changing the start-up 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 start-up routine the initial processing of the external variable to be stored to the new section.

To the start-up routine, therefore, add the declaration of the first label of the new section and the portion that copies the initial value, and add the portion that declares the end label to the 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 start-up routine cstartx.asm)

<1> Add the declaration of the label indicating the end of the section with the changed name

```
                .
                .
                .
        EXTRN   _main,_exit,_@STBEG
        EXTRN   _?R_INIT,_?R_INIS,_?DATA,_?DATS

        EXTRN   RTT1_E,TT1_E     ←  Adds EXTRN declaration of RTT1_E and TT1_E
                .
                .
                .
```

#pragma section… #pragma section…

<2> Add a section to copy the initial values from the RTT1 section with the changed name to the TT1 section.

```
            .
            .
            .
LDATS1:
            MOVW        AX,HL
            CMPW        AX,#_?DATS
            BZ          $LDATS2
            MOV         A,#0
            MOV         [HL],A
            INCW        HL
            BR          $LDATS1
LDATS2:
            MOVW        DE,#TT1_S
            MOVW        HL,#RTT1_S
LTT1:
            MOVW        AX,HL
            CMPW        AX,#RTT1_E
            BZ          $LTT2
            MOV         A,[HL]
            MOV         [DE],A
            INCW        HL
            INCW        DE
            BR          $LTT1
LTT2:
;
            CALL        !_main          ; main();
            MOVW        AX,#0
            CALL        !_exit          ; exit(0);
            BR          $$
;
```

Adds section to copy the initial values from the RTT1 section to the TT1 section (LDATS2 through BR $LTT1)

#pragma section… **#pragma section…**

<3> Set the label of the start of the section with the changed name.

```
             .
             .
             .
@@R_INIT     CSEG       UNITP
_@R_INIT:
@@R_INIS     CSEG       UNITP
_@R_INIS:
@@INIT       DSEG       UNITP
_@INIT:
@@DATA       DSEG       UNITP
_@DATA:
@@INIS       DSEG       SADDRP
_@INIS:
@@DATS       DSEG       SADDRP
_@DATS:                                 ; Indicates the start of the RTT1 section
                                        ; Adds the label setting
RTT1         CSEG       UNITP           ; Indicates the start of the TT1 section
RTT1_S:                                 ; Adds the label setting
TT1          DSEG       UNITP
TT1_S:

@@CALT       CSEG       CALLT0
@@CALF       CSEG       FIXED
@@CNST       CSEG       UNITP
@@BITS       BSEG
;
             END
```

#pragma section… #pragma section…

(Changing end module rom.asm)

(1) Add the declaration of the label indicating the end of the section with the changed name

```
NAME        @rom
;
PUBLIC      _?R_INIT,_?R_INIS
PUBLIC      _?INIT,_?DATA,_?INIS,_?DATS

PUBLIC      RTT1_E,TT1_E    ←      Adds RTT1_E and TT1_E

;
@@R_INIT    CSEG       UNITP
_?R_INIT:
@@R_INIS    CSEG       UNITP
_?R_INIS:
@@INIT      DSEG       UNITP
_?INIT:
@@DATA      DSEG       UNITP
_?DATA:
@@INIS      DSEG       SADDRP
_?INIS:
@@DATS      DSEG       SADDRP
_?DATS
    .
    .
    .
```

(2) Setting the label indicating the end

```
    .
    .
    .
RTT1        CSEG       UNITP    ; Adds the label setting indicating the end of the RTT1 section.
RTT1_E:                         ; Adds the label setting

TT1         DSEG       UNITP    ; Adds the label setting indicating the end of the TT1 section.
TT1_E:                          ; Adds the label setting

;
            END
```

(17) Binary constant

Binary Constant — Binary constant 0bxxx

FUNCTION

- Describes binary constants to the location where integer constants can be described.

EFFECT

- Constants can be described in bit strings without being replaced with octal or hexadecimal number. Readability is also improved.

USAGE

- Describe binary constants in the C source. The following shows the description method of binary constants.

```
0b binary number
0B binary number
```

Remark Binary number: either '0' or '1'

- A binary constant has 0b or 0B at the start and is followed by the list of numbers 0 or 1.
- The value of a binary constant is calculated with 2 as the base.
- The type of a binary constant is the first one that can express the value in the following list.
  - Subscripted binary number : **int**, **unsigned int**, **long int** **unsigned long int**
  - Subscripted u or U : **unsigned int**, **unsigned long int**
  - Subscripted l or L : **long int** **unsigned long int**
  - Subscripted u or U and subscripted l or L with : **unsigned long int**

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>

- Modifications are needed to meet the specification of the compiler if the compiler supports binary constants.
- Modifications into other integer formats such as octal, decimal, and hexadecimal are needed if the compiler does not support binary constants.

(18) 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 a 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 **NAME** pseudo instruction when **-NG** is specified.
- If a module name with nine or more letters are 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 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/lowercase is distinguished.

EXAMPLE

```
#pragma name module1
   .
   .
   .
```

COMPATIBILITY

<From another C compiler to this C compiler>

- Modifications are not needed if the compiler does not support the module name changing function.
- To change a module name, modification is made according to **USAGE** above.

<From this C compiler to another C compiler>

- **#pragma** name . . . is deleted or sorted by **#ifdef**.
- To change a module name, modification is needed depending on the specification of each compiler.

(19) 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

- Rotate function is realized by the C source or **ASM** description without describing the processing to perform rotate.

USAGE

- Describe in the source in the same format as the function call. There are the following 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 right for y times.

(b)
```
unsigned char rolb (x, y) ;
unsigned char x ;
unsigned char y ;
```

Rotates x to left for y times.

(c)
```
unsigned int rorw (x, y) ;
unsigned int x ;
unsigned char y ;
```

Rotates x to right for y times.

(d)
```
unsigned int rolw (x, y)
unsigned int x ;
unsigned char y ;
```

Rotates x to left for y times.

Caution The above mentioned function declaration is not affected by the -ZI option.

Rotate Function #pragma rot

- Declare the use of the function for rotate by the **#pragma rot** directive of the module.
  However, the followings can be described before **#pragma rot**.
  - Comments
  - Other **#pragma** directives
  - Preprocessing directives which do not generate definition/reference of variables and definition/reference of functions
- Keywords following **#pragma** can be described in either uppercase or lowercase letters.

EXAMPLE

(C source)

```
#pragma rot
unsigned char a = 0x11 ;
unsigned char b = 2 ;
unsigned char c ;
void main ( ) {
   c = rorb(a, b) ;
}
```

(Output assembler source)

```
        mov    a,!_b
        mov    c,a
        mov    a,!_a
        ror    a, 1
        dbnz   c,$$-1
        mov    !_c,a
```

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.

Rotate Function #pragma rot

COMPATIBILITY

<From another C compiler to this C compiler>

- Modification is not needed if the compiler does not use the functions for rotate.
- To change to functions for rotate, modifications are made according to **USAGE** above.

<From this C compiler to another C compiler>

- **#pragma rot** statement is deleted or sorted by **#ifdef**.
- To use as a function for rotate, modification is needed depending on the specification of each compiler (**#asm**, **#endasm** or **asm()** ; , etc.).

(20) Multiplication function

Multiplication Function #pragma mul

FUNCTION

- Outputs the code that multiplies the value of an expression to the object with direct inline expansion instead of function call and generates an object file.
- If there is not a **#pragma** directive, the multiplication function is regarded as an ordinary function.

EFFECT

- The codes utilizing the data size of input/output of the multiplication instruction are generated. Therefore, the codes with faster execution speed and smaller size than the description of ordinary multiplication expressions can be generated.

USAGE

- Describe in the same format as that of function call in the source.

```
mulu
```

[List of multiplication function]

```
unsigned int mulu (x, y) ;
unsigned char x ;
unsigned char y ;
```

Performs unsigned multiplication of x and y.

- Declare the use of functions for multiplication by **#pragma** mul directive of the module.
  However, the followings can be described before #pragma mul.
  - Comments
  - Other **#pragma** directives
  - Preprocessing directives that do not generate definition/reference of variables and definition/reference of functions
- Keywords following **#pragma** can be described in either uppercase or lowercase letters.

Multiplication Function #pragma mul

RESTRICTIONS

- Multiplication functions are called by the library, if the target device does not have multiplication instructions.
- The function for multiplication cannot be used as the function names (when **#pragma mul** is declared).
- The function for multiplication must be described in lowercase letters. If they are described in uppercase letters, they are handled as ordinary functions.

EXAMPLE

(C source)

```
#pragma mul
unsigned char a = 0x11 ;
unsigned char b = 2 ;
unsigned int i ;
void main()
{
        i = mulu(a, b) ;
}
```

(Output object of compiler)

```
mov     a,!_b
mov     x,a
mov     a,!_a
mulu    x
movw    !_i,ax
```

COMPATIBILITY

<From another C compiler to this C compiler>

- Modifications are not needed if the compiler does not use the functions for multiplication.
- To change to functions for multiplication, modification is made according to **USAGE** above.

<From this C compiler to another C compiler>

- **#pragma mul** statement is deleted or sorted by #ifdef. Function names for multiplication can be used as the function names.
- To use as functions for multiplication, modification is needed depending on the specification of each compiler (**#asm**, **#endasm** or **asm()** ;, etc.).

(21) Division function

Division Function #pragma div

FUNCTION

- Outputs the code which divides the value of an expression from 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

- The codes utilizing the data size of input/output of the division instruction are generated. Therefore, the codes with faster execution speed and smaller size than the description of ordinary division expressions can be generated.

USAGE

- Describe in the same format as that of function call in the source. There are the following two functions for division.

```
divuw, moduw
```

List of division function

(a)
```
unsigned int divuw(x, y) ;
unsigned int x ;
unsigned char y ;
```

Performs unsigned division of x and y and returns the quotient.

(b)
```
unsigned char moduw(x, y) ;
unsigned int x ;
unsigned char y ;
```

Performs unsigned division of x and y and returns the remainder.

Caution The above mentioned function declaration is not affected by the -ZI option.

- Declare the use of the function for divisions by the **#pragma div** directive of the module.
  However, the followings can be described before **#pragma div**.
  - Comments
  - Other **#pragma** directives
  - Preprocessing directives which do not generate definition/reference of variables and definition/reference of functions
- Keywords following **#pragma** can be described in either uppercase or lowercase letters.

Division Function #pragma div

RESTRICTIONS

- The division function is called by the library if the target device does not have division instruction.
- 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)

```
        mov     a,!_b
        mov     c,a
        movw    ax,!_a
        divuw   c
        movw    !_i,ax
        mov     a,!_b
        mov     c,a
        movw    ax,!_a
        divuw   c
        mov     a,c
        mov     !_c,a
```

Division Function #pragma div

COMPATIBILITY

<From another C compiler to this C compiler>

- Modification is not needed if the compiler does not use the functions for division.
- To change to functions for division, modifications are made according to **USAGE** above.

<From this C compiler to another C compiler>

- **#pragma div** statement is deleted or sorted by **#ifdef**. The function names for division can be used as the function name.
- To use as a function for division, modification is needed depending on the specification of each compiler (**#asm**, **#endasm** or **asm()** ; , etc.).

(22) BCD operation function

BCD Operation Function #pragma bcd

FUNCTION

- Outputs the code that performs a BCD operation on the expression value in an object by direct inline expansion rather than by function call, and generates an object file.
- If there are no **#pragma** directives, the function for BCD operation is regarded as an ordinary function.

EFFECT

- Even if the process of the BCD operation is not described, the BCD operation function can be realized by the C source or ASM statements.

USAGE

- The same format as that of a function call is coded in the source. There are 13 types of function name for BCD operation, as listed below. Refer to **List of functions for BCD operation**, later in this chapter for more information.

```
adbcdb, sbbcdb, adbcdbe, sbbcdbe, adbcdw, sbbcdw, adbcdwe,
sbbcdwe, bcdtob, btobcde, bcdtow, wtobcd, btobcd
```

- Use of functions for division is declared by the module's #pragma bcd directive. The following items, however, can be coded before #pragma bcd.
  - Comments
  - Other **#pragma** directives
  - Preprocessing directives that do not generate definitions/references of variables or function definitions/references
- Either uppercase or lowercase letters can be used for keywords described after **#pragma**.

RESTRICTIONS

- **BCD** operation function names cannot be used as function names.
- The **BCD** operation function is coded in lowercase letters. If uppercase letters are used, these functions are regarded as an ordinary functions.
- The adbcdwe and sbbcdwe are not supported in the static model.

BCD Operation Function #pragma bcd

EXAMPLE

(C source)

```
#pragma bcd
unsigned char a = 0x12 ;
unsigned char b = 0x34 ;
unsigned char c ;
void main ( )
{
    c = adbcdb (a, b) ;
    c = sbbcdb (b, a) ;
}
```

(Output assembler source)

```
mov     a, !_a
add     a, !_b
adjba
mov     !_c, a
mov     a, !_b
sub     a, !_a
adjbs
mov     !_c, a
```

[List of functions for BCD operation]

(1) **`unsigned char adbcdb (x, y) ;`**
`unsigned char x ;`
`unsigned char y ;`
Decimal addition is carried out by the **BCD** adjustment instruction.

(2) **`unsigned char sbbcdb (x, y) ;`**
`unsigned char x ;`
`unsigned char y ;`
Decimal subtraction is carried out by the **BCD** adjustment instruction.

BCD Operation Function #pragma bcd

(3) `unsigned int adbcdbe (x, y) ;`
`unsigned char x ;`
`unsigned char y ;`
Decimal addition is carried out by the **BCD** adjustment instruction (with result expansion).

(4) `unsigned int sbbcdbe (x, y) ;`
`unsigned char x ;`
`unsigned char y ;`
Decimal subtraction is carried out by the **BCD** adjustment instruction (with result expansion). If a borrow occurs, the high-order digits are set to 0x99.

(5) `unsigned int adbcdw (x, y) ;`
`unsigned int x ;`
`unsigned int y ;`
Decimal addition is carried out by the **BCD** adjustment instruction.

(6) `unsigned int sbbcdw (x, y) ;`
`unsigned int x ;`
`unsigned int y ;`
Decimal subtraction is carried out by the **BCD** adjustment instruction.

(7) `unsigned long adbcdwe (x, y) ;`
`unsigned int x ;`
`unsigned int y ;`
Decimal addition is carried out by the **BCD** adjustment instruction (with result expansion).

(8) `unsigned long sbbcdwe (x, y) ;`
`unsigned int x ;`
`unsigned int y ;`
Decimal subtraction is carried out by the **BCD** adjustment instruction (with result expansion). If a borrow is occurred, the higher digits are set to 0x9999.

(9) `unsigned char bcdtob (x) .`
`unsigned char x ;`
Values in decimal number are converted to binary number values.

(10) `unsigned int btobcde (x) ;`
`unsigned char x ;`
Values in binary number are converted to decimal number values.

BCD Operation Function #pragma bcd

(11) `unsigned int bcdtow (x) ;`

`unsigned int x ;`

Values in decimal number are converted to binary number values.

(12) `unsigned int wtobcd (x) ;`

`unsigned int x ;`

Values in decimal number are converted to binary number values. However, if the value of x exceeds 10000, 0xffff is returned.

(13) `unsigned char btobcd (x) ;`

`unsigned char x ;`

Values in decimal number are converted to those in binary number. However, the overflow is discarded.

Caution The above-mentioned function declarations are not influenced by the -ZI and -ZL options.

COMPATIBILITY

<From another C compiler to this C compiler>

- Corrections are not needed if functions for the **BCD** operations are not used.
- To change another function to the function for **BCD** operation, use the description above.

<From this C compiler to another C compiler>

- The **#pragma bcd** statements are either deleted or separated by **#ifdef**. A BCD operation function name can be used as a function name.
- If using "**pragma bcd**" as a BCD operation function, the changes to the program source must conform to the C compiler's specifications **(#asm**, **#endasm** or **asm()**; etc.).

(23) Bank function

Bank Function _ _banked1 to _ _banked15

FUNCTION

- Stores the function body in bank area, and calls function via a library for bank function call.
- To call functions declared as **_ _banked1 to _ _banked15** (referred to as bank function), the name with **_** at the beginning of function name should be used, as in ordinary functions.

EFFECT

- Can be located at code block exceeding 64 KB.

USAGE

- Attributes of **_ _banked1** to **_ _banked15** are added at the beginning during function declaration.

EXAMPLE

(C source)

```
========== cb1.c ==========
_ _banked1 extern int tsub () ;
void main ()
{
        int ret_val ;
        ret_val = tsub () ;
}
```

```
========== cb2.c ==========
_ _banked1 int tsub ()
{
        int val = 1 ;
        return val ;
}
```

Bank Function _ _banked1 to _ _banked15

(Output assembler source)

cb1 module

```
            EXTRN    _tsub                 ; Declaration
            EXTRN    @@fcall
            PUBLIC   _main

    @@CODE CSEG
    _main :
            push     hl
            push     hl                    ;
            movw     hl,#_tsub             ; Function name
            mov      e,#01H                ; Bank number
            callt    [@@fcall]             ; Call
            pop      hl                    ;
            movw     ax, bc
            movw     hl, ax
            pop      hl
            ret
```

cb2 module

```
             PUBLIC   _tsub                ; Declaration

    @@BANK1  CSEG
    _tsub :                                ; Function definition
             push     hl
             movw     hl,#01H
             movw     ax, hl
             movw     bc, ax
             pop      hl
             ret
```

The segment in which the function main body belongs becomes **@@BANK1 CSEG** to **@@BANK15 CSEG** according to the declaration of **_ _banked1** to **_ _banked15**.

Bank Function _ _banked1 to _ _banked15

The routine that is supplied by the compiler is as follows.

```
          NAME    @fcall

          PUBLIC  @@fcall

PORT      EQU     P0                        ; Ports are changed according to the systems

@@CALT    CSEG    CALLT0
@@fcall  :DW      ?@fcall

@@CODE    CSEG
?@fcall  :
          MOV     D, A                      ; Save A register
          MOV     A, PORT                   ; Obtain the current bank name
          PUSH    AX                        ; Saves the current bank name
          MOV     A, E
          MOV     PORT, A                   ; Set new bank name
          MOV     A, D                      ; Restore A register
          MOVW    DE,#FAR_RET
          PUSH    DE
          PUSH    HL
          RET
FAR_RET  :
          POP     AX
          MOV     PORT, A                   ; Restore the saved bank name.
          RET
```

CAUTIONS

- Ports can be changed to R/W-enabled ports, which include other latch functions, according to the user's environment.
- Since a bank switching function are not provided in this C compiler, a circuit to switch banks is required. More specifically, as the bank numbers from 1 to 15 are output at the ports, a decoder to select the ROM that corresponds to these values should be prepared.

Bank Function _ _banked1 to _ _banked15

RESTRICTIONS

- Only the function body is allocated to the bank area. Data cannot be allocated to the bank area.
- The banks able to be used by the function definition are restricted to one bank per file.
- A pointer for the bank function cannot be used. A table jump can be realized by using the Table jump function.
- **callt**/**callf**/**noauto**/**norec**/**_ _callt**/**_ _callf**/**_ _leaf**/**_ _interrupt**/**_ _interrupt_brk**/**_ _rtos_interrupt**/**_ _pascal**/**_ _flash**/**_ flashf** cannot be specified as a bank function.
- The bank function is not supported when the static model is specified. A warning message is output to the place where **_ _banked 1** to **_ _banked 15** keywords appeared at first, the keywords are omitted, and the bank function is processed as an ordinary function.

COMPATIBILITY

<From another C compiler to this C compiler>

- Modification is not needed unless **_ _banked1** to **_ _banked15** are used as reserved words.
- Modify in accordance with the above method when changing to bank functions.

<From this C compiler to another C compiler>

- Compatibility is maintained by using **#define** (Refer to **11.6 Modifications of C Source**).
- By this modification, bank functions are treated as an ordinary function.

(24) Table jump function

Table Jump Function #pragma tbl

FUNCTION

- Generates the code that calls the function that corresponds to the values of an expression. It does not create function table but creates codes that select several function call processes.
- Prototype declaration defined in functions is applied to the prototype declaration for arguments.
- If there is no **#pragma** directive, the function for table jump is regarded as an ordinary function.

EFFECT

- Permits a table jump using a combination of different function attributes.
- Permits a table jump to the bank function.

USAGE

- This function is described in the source in the same format as a function call. There are 8 types of table jump function names, as listed below. Refer to **List of functions for table jump**, later in this chapter for more information.

```
vtbljp, ctbljp, uctbljp, itbljp, uitbljp, ltbljp, ultbljp, vptbljp
```

- Use of functions for Table jump is declared by the **#pragma tbl** directive of modules. The following items, however, can be described before **#pragma tbl**.
  - Comments
  - Other **#pragma** directive
  - Preprocessing directives which do not generate definition/reference of variables and definitions/references of functions
- Either uppercase or lowercase letters can be used for keywords described following **#pragma**.

EXAMPLE

(C source)

```
#pragma tbl
int func1 (int x, int y) ;
_ _banked1 int func2 (int x, int y) ;
unsigned char c ;
int i ;
void main ()
{
        i = itbljp (c, 2, func1, func2, 1, 2) ;
}
```

Table Jump Function #pragma tbl

(Output assembler source)

```
_main :
        mov     a,!_c
        cmp     a,#00H
        bnz     $?L0004
        movw    ax,#02H                  ;
        push    ax                       ;
        decw    ax                       ;
        call    !_func1                  ; func1 call processing
        pop     ax                       ;
        br      $?L0005
?L0004 :
        cmp     a,#01H
        bnz     $?L0005
        movw    ax,#02H                  ;
        push    ax                       ;
        decw    ax                       ;
        push    hl                       ;
        movw    hl,#_func2               ; func2 call processing
        mov     e,#01H                   ;
        callt   [@@fcall]                ;
        pop     hl                       ;
        pop     ax                       ;
?L0005 :
        movw    ax, bc
        movw    !_i, ax
        ret
```

Table Jump Function #pragma tbl

[List of functions for table jump]

(1) **void vtbljp(unsigned char** x,**int** n,function name 1,…function name n,argument 1,…argument m);
The x+1th function is called by using m arguments. There are no return value. n is the number of described function names written with a constant.

(2) **char ctbljp(unsigned char** x,**int** n,function name 1,…function name n,argument 1,…argument m);
The x+1th function is called by using m arguments. The return value is **char** type. n is the number of described function names written with a constant.

(3) **unsigned char uctbljp(unsigned char** x,**int** n,function name 1,…function name n,argument 1,…argument m);
The x+1th function is called by using m arguments. The return value is unsigned **char** type. n is the number of described function names written with a constant.

(4) **int itbljp(unsigned char** x,**int** n, function name 1,…function name n,argument 1,…argument m);
The x + 1th function is called by using m arguments. The return value is **int** type. n is the number of described function names written with a constant.

(5) **unsigned int uitbljp(unsigned char** x,**int** n,function name 1,…function name n,argument 1,…argument m);
The x + 1th function is called by using m arguments. The return value is unsigned **int** type. n is the number of described function names written with a constant.

(6) **long ltbljp(unsigned char** x,**int** n,function name 1,…function name n,argument 1,…argument m);
The x + 1th function is called by using m arguments. The return value is unsigned **long** type. n is the number of described function names written with a constant.

(7) **unsigned long ultbljp(unsigned char** x,**int** n,function name 1,…function name n,argument 1,…argument m);
The x+1th function is called by using m arguments. The return value is **unsigned long** type. n is the number of described function names written with a constant.

(8) **void * vptbljp(unsigned char** x,**int** n,function name 1,…function name n,argument 1,…argument m);
The x + 1th function is called by using m arguments. The return value is **void * type**. n is the number of described function names written with a constant.

Caution The above mentioned function declaration is not affected by the -ZI and -ZL options.

Table Jump Function #pragma tbl

RESTRICTIONS

- Table jump function names cannot be used as function names.
- The number of arguments must be same between call functions.
- Table jump functions should be described in lowercase letters. If uppercase letters are used, this function is regarded as an ordinary function.

COMPATIBILITY

<From another C compiler to this C compiler>

- Modification is not needed if functions for table jump are not used.
- To change another function to the function for Table jump, use the method above.

<From this C compiler to another C compiler>

- Delete the “**#pragma tbl**” statement or delimit it by **#ifdef**. Table jump function names can be used as function names.
- When using “**#pragma tbl**” as a table jump function, the changes to the program source must conform to the C compiler specifications.

(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 instruction can be embedded in the code area without using the **ASM** statement.
  When **ASM** is used, an object cannot be obtained without the intermediary of assembler. On the other hand, if the data insertion function is used, an object can be obtained without the intermediary of assembler.

USAGE

- Describe using uppercase letters in the source in the same format as that of function call.
- The function name for data insertion is **_ _OPC**.

[List of data insertion functions]

```
void _ _OPC (unsigned char x,…);
```

Insert the value of the constant described in the argument to the current address.
Arguments can describe only constants.

- Declare the use of functions for data insertion by the **#pragma** opc directive.
  However, the followings can be described before **#pragma** opc.
  - Comments
  - Other **#pragma** directives
  - Preprocessing directives which do not generate definition/reference of variables and definition/reference of functions
- Keywords following **#pragma** can be described in either uppercase or lowercase letters.

RESTRICTIONS

- The function names for data insertion cannot be used as the function names (when **#opc** is specified).
- **_ _OPC** must be described in uppercase letters. If they are described in lowercase letters, they are handled as ordinary functions.

Data Insertion Function #pragma opc

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
; line   7 : }
    ret
```

COMPATIBILITY

<From another C compiler to this C compiler>

- Modification is not needed if the compiler does not use the functions for data insertion.
- To change to functions for data insertion, use the **USAGE** above.

<From this C compiler to another C compiler>

- The **#pragma opc** statement is deleted or delimited by **#ifdef**. Function names for data insertion can be used as function names.
- To use as a function for data insertion, changes to the program source must conform to the specification of the C compiler (**#asm**, **#endasm** or **asm()** ; , etc.).

(26) Interrupt handler for real-time OS (RTOS)

Interrupt Handler for RTOS #pragma rtos_interrupt ...

FUNCTION

- Interprets the function name specified with the **#pragma rtos_interrupt** directive as the interrupt handler for the 78K0 Series RTOS (real-time OS) RX78K0.
- Registers the address of the described function name to the interrupt vector table for the specified interrupt request name.
- When the 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 two system call calling functions **ret_int/ret_wup** can be called in the interrupt handler for RTOS (for the details of the RTOS system call calling function, refer to the **List of RTOS system call calling function** 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 with unconditional branch instruction.
- If there is neither a **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) Saves the **saddr** area used by compiler
(3) Changes the stack pointer (only when the stack change is specified)
(4) Secures the local variable area (only when there is a local variable)
(5) The function body
(6) Releases the local variable area (only when there is a local variable)
(7) Sets back the stack pointer (only when the stack change is specified)
(8) Restores the **saddr** area used by compiler
(9) Restores all the registers
(10) RETI

For the **ret_int/ret_wup** described in the middle of the function, the codes in (6) and (7) are generated immediately before the unconditional branch instruction each time.
If a function ends with **ret_int/ret_wup**, the codes in (8) through (10) are not generated.

EFFECT

- The interrupt handler for RTOS can be described in the C source level.
- Because the interrupt request name is identified, the address of the vector table does not need to be identified.

Interrupt Handler for RTOS #pragma rtos_interrupt ...

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, main **#pragma** directive is described after #pragma pc.
  The following can be described before the **#pragma** directive.
  - Comments
  - Preprocessing directives which do not generate definition/reference of variables and definition/reference of functions

```
#pragmaΔrtos_interrupt[ΔInterrupt request nameΔfunction nameΔ[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 function]

(1) `void ret_int ( ) ;`
Calls RTOS system call ret_int.

(2) `void ret_wup (x) ;`
`unsigned char *x ;`

Calls RTOS system call **ret_wup** with x as an argument.

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.
- Interrupt requests are double-checked in one module units only.
- In the case an interrupt (the same or another interrupt) is generated duplicatedly during vector interrupt processing depending on the contents of the priority specification flag register, interrupt mask flag register, etc., if the stack change is specified, the contents of the stack is updated, which may cause troubles. However, the compiler cannot check, so that care must be taken.
- The interrupt handler for RTOS cannot specify **callt/callf/noauto/norec/_ _callt/_ _callf/_ _leaf/_ _interrupt/_ _interrupt_brk/_ _banked1** to **15/_ _pascal/_ _flash/_ _flashf**.
  The RTOS system call calling function names **ret_int/ret_wup** cannot be used for the function names.
  If the functions which specified the stack change by **#pragma rtos_interrupt** specifications 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.

Interrupt Handler for RTOS #pragma rtos_interrupt ...

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 () ;
        }
}
```

(Output object of compiler)

```
@@CODE  CSEG
_intp :
        push    ax                          ; Saves registers
        push    bc                          ;
        push    de                          ;
        push    hl                          ;
        movw    ax,_@RTARG0                 ; Saves saddr area used the compiler
        push    ax                          ;
        movw    ax,_@RTARG2                 ;
        push    ax                          ;
        movw    ax,_@RTARG4                 ;
        push    ax                          ;
        movw    ax,_@RTARG6                 ;
        push    ax                          ;
        movw    hl,#01H ; 1
        movw    ax,!_i
        cmpw    ax,#01H ; 1
        bnz     $?L0003
        br      !_ret_int
```

Interrupt Handler for RTOS #pragma rtos_interrupt ...

(Output object of compiler …continued)

```
?L0003 :
          pop     ax                        ; Restores saddr area used by the compiler
          movw    _@RTARG6, ax              ;
          pop     ax                        ;
          movw    _@RTARG4, ax              ;
          pop     ax                        ;
          movw    _@RTARG2, ax              ;
          pop     ax                        ;
         movw    _@RTARG0, ax               ;
         pop     hl                         ; Restores registers
         pop     de                         ;
         pop     bc                         ;
         pop     ax                         ;
         reti
@@VECT06 CSEG    AT            0006H
_@vect06 :
          DW      _intp
```

(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 char  TaskID1 ;
void intp () {
       int a ;
       a = 1 ;
       if (i = = 1) {
            ret_wup (&TaskID1) ;
       }
}
```

Interrupt Handler for RTOS #pragma rtos_interrupt ...

(Output object of compiler)

```
_intp :
        push    ax                      ; Saves registers
        push    bc                      ;
        push    de                      ;
        push    hl                      ;
        movw    ax,_@RTARG0             ; Saves saddr area used by the compiler
        push    ax                      ;
        movw    ax,_@RTARG2             ;
        push    ax                      ;
        movw    ax,_@RTARG4             ;
        push    ax                      ;
        movw    ax,_@RTARG6             ;
        push    ax                      ;
        movw    ax, sp                  ; Switch stack pointer
        movw    sp,#_buff + 10          ;
        push    ax                      ;
        movw    hl,#01H ; 1
        movw    ax,!_i
        cmpw    ax,#01H ; 1
        bnz     $?L0003
        movw    hl,#_TaskID1
        pop     ax                      ; Returns stack pointer
        movw    sp, ax                  ;
        br      !_ret_wup
?L0003 :
        pop     ax                      ; Returns stack pointer
        movw    sp, ax                  ;
        pop     ax                      ; Restores saddr area used by the compiler
        movw    _@RTARG6, ax            ;
        pop     ax                      ;
        movw    _@RTARG4, ax            ;
        pop     ax                      ;
        movw    _@RTARG2, ax            ;
        pop     ax                      ;
        movw    _@RTARG0, ax            ;
        pop     hl                      ; Restores registers
        pop     de                      ;
        pop     bc                      ;
        pop     ax                      ;
        reti
@@VECT06 CSEG   AT              0006H
_@vect06 :
        DW      _intp
```

Interrupt Handler for RTOS #pragma rtos_interrupt ...

COMPATIBILITY

<From another C compiler to this C compiler>

- Modifications are not needed if the compiler does not support the interrupt handler for RTOS.
- To change to interrupt handler for RTOS, use the **USAGE** above.

<From this C compiler to another C compiler>

- Handled as an ordinary function if **#pragma rtos_interrupt** specification is deleted.
- To use as an interrupt handler for ROTS, changes to the source program must conform to the specification of the C compiler.

(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 keywords **_ _rtos_interrupt** (for the details of the RTOS system call calling function, refer to the **List of RTOS system call calling function** above).
  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 function **ret_int/ret_wup** are called with an unconditional branch instruction.
- If there is neither a **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 RTOS system call calling function]

(a) `void ret_int ( ) ;`

Calls system call **ret_int** for RTOS.

(b) `void ret_wup (x) ;`
`unsigned char *x ;`

Calls system call **ret_wup** for RTOS with x as an argument.

Interrupt Handler Qualifier for RTOS _ _rtos_interrupt

RESTRICTIONS

- The interrupt handler for RTOS cannot specify **callt/callf/noauto/norec/_ _callt/_ _callf/_ _leaf/_ _interrupt/ _ _interrupt_brk/_ _banked1** to **15/_ _pascal/_ _flash/_ _flashf**.
- RTOS system call calling function name **ret_int/ret_wup** cannot be used for the function names.
- The **_ _rtos_interrupt** modifier is not supported when the static model is specified. A warning message is output to the place where **_ _rtos_interrupt** first appeared, **_ _rtos_interrupt** is omitted, and it is processed as a normal function.

CAUTIONS

- Vector addresses cannot be set only with declaration of this qualifier.
  The setting of the vector address must be performed separately with the **#pragma** directive, assembler description, etc.
- When the interrupt handler for RTOS is defined in the same file as the one in which the **#pragma rtos_interrupt** ··· is specified, the function name specified with **#pragma rtos_interrupt** is judged as an interrupt handler for RTOS even if this qualifier is not described.

COMPATIBILITY

<From another C compiler to this C compiler>

- Modifications are not needed if the compiler does not support interrupt handler for RTOS.
- To change to interrupt handler for RTOS, use the **USAGE** above.

<From this C compiler to another C compiler>

- Changes can be made by **#define** (For the details, refer to **11.6 Modifications of C Source**). By these changes, interrupt handler qualifiers for RTOS are handled as ordinary variables.
- To use as an interrupt handler for RTOS, modification is needed depending on the specification of each compiler.

(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.
- In the case the function name is specified, if 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 function can be used.

[**RTOS system call calling function]**

(a) void ext_tsk (void) ;

Calls RTOS system call **ext_tsk**.

When **ext_tsk** is, however, called in the **ext_tsk** prototype declaration and entity definition, interrupt function, interrupt handler for RTOS, an error occurs.

- The RTOS system call calling function of **ext_tsk** is called with an unconditional branch instruction. If **ext_tsk** is issued after function, the postprocessing is not output.
- When there is no **ext_tsk** in the task function for RTOS and **-W2** option is specified, a warning message is output.

EFFECT

- The task function for RTOS can be described in the C source level.
- The saving and postprocessing of the register frame pointer/register variable are not output, so the code efficiency is improved.

USAGE

- Specifies the function name for the following **#pragma** directives.

```
#pragmaΔrtos_task[Δtask-function-name]
```

- The **#pragma** directives are described at the start of the C source.
  However, the followings can be described before the **#pragma** directive.
  - Comments
  - Preprocessing directives which do not generate definition/reference of variables and definition/reference of functions
- Keywords following **#pragma** can be described either in uppercase or lowercase letters.

Task Function for RTOS #pragma rtos_task

RESTRICTIONS

- The task function for RTOS cannot specify the **callt/callf/noauto/norec/_ _callt/_ _callf/_ _leaf/_ _interrupt/_ _interrupt_brk/_ _rtos_interrupt/_ _banked1** to **15/_ _pascal/_ _flash/_ _flashf**.
- The task function for RTOS cannot be called in the same manner as the ordinary functions.
  RTOS system call calling function name **ext_tsk** cannot be used for function names.
- The task function for RTOS is not supported when the static model is specified.

EXAMPLE

(C source)

```
#pragma rtos_task func
int i ;
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)

```
@@CODE   CSEG
_main :
         push    hl
         movw    hl,#01H ; 1
         br      !_ext_tsk               ; Epilogue is not output
_func :
         push    ax
         push    ax                      ; Frame pointers are not saved
         movw    ax, sp
         movw    hl, ax
         movw    ax,#01H ; 1
         mov     [hl + 1], a             ; x
         xch     a, x
         mov     [hl], a                 ; x
         movw    ax,#02H ; 2
         mov     [hl + 3], a             ; r
         xch     a, x
         mov     [hl + 2], a             ; r
         br      !_ext_tsk               ; Epilogue is not output
```

COMPATIBILITY

<From another C compiler to this C compiler>

- Modifications are not needed if the compiler does not support the task function for RTOS.
- To change to the task function for RTOS, use the **USAGE** above.

<From this C compiler to another C compiler>

- If **#pragma rtos_task** specification is deleted, RTOS task function is used as an ordinary function.
- To use as RTOS task function, changes to the program source must conform to the specification of the C compiler.

(29) Static model

Static Model

FUNCTION

- All arguments are passed through registers (Refer to **11.7.5 Static model function call interface**).
- Function arguments that are passed through registers are allocated in the function-specific static area.
- Automatic variables are allocated to the function-specific static area.
- In the case of the **leaf** function[Note], arguments and automatic variables are allocated to the saddr area below 0FEDFH, in the order of description starting from the high-order addresses. Since the **saddr** area is commonly used by the **leaf** functions of all modules, this area is referred to as the shared area. The maximum size of the shared area is defined by the parameter when the **-SM** option is specified.

```
-SM [nn]: nn = 0 to 16
```

nn bytes are assigned as shared area and the rest are allocated to the function-specific static area. If nn = 00 is specified or this specification is omitted, the shared area is not used.

Note For the functions that do not call functions, it is not necessary to describe **norec/_ _leaf** since the compiler executes automatic determination.

- It is possible to add the **sreg/_ _sreg** keywords to function arguments and automatic variables. Function arguments and automatic variables that have the **sreg/_ _sreg** keywords added are allocated to the **saddr** area. As a result, bit manipulation becomes possible.
- By specifying the **-RK** option, function arguments and automatic variables (except for the static variables in functions) are allocated to **saddr** and bit manipulation becomes possible (Refer to **11.5 (3) How to use the saddr area**).
- The compiler executes the following macro definition automatically.

```
#define _ _STATIC_MODEL_ _ 1
```

EFFECT

- Normally, instructions that access the static area are shorter and faster than those that access static frames. Accordingly, it is possible to shorten object codes and increase execution speed.
- The save/restore processing of arguments and variables that use the **saddr** area (register variables in interrupt functions, **norec** function argument/automatic variables, run time library augments) is not performed, as a result, it is possible to increase the speed of interrupt processing.
- Memory space can be saved since the data area is commonly used by several **leaf** functions.

USAGE

- Specify the **-SM** option during compilation.
  The object in this case is called the static model, while the object without specification of the **-SM** option is called normal model.

Static Model

EXAMPLE

An example of the **-SM4** specification is as follows.

(C source)

```
void sub (char, char, char) ;
void main ()
{
        char i = 1 ;
        char j, k ;
        j = 2 ;
        k = i + j ;
        sub (i, j, k) ;
}
void sub (char p1, char p2, char p3)
{
        char a1, a2 ;
        a1 = 1<<p1 ;
        a2 = p2 + p3 ;
}
```

(Output object of compiler)

```
@@DATA  DSEG  UNITP
L0003:  DS    (1)                         ; Automatic variable k of function main
        DS    (1)

; line   1: void sub (char, char, char) ;
; line   2: void main ()
; line   3: {

@@CODE  CSEG
_main:
        push   de
; line   4 : char i = 1 ;
        mov    e,#01H ; 1                 ; Automatic variable i
; line   5:  char j, k ;
; line   6:  j = 2 ;
        mov    d,#02H ; 2                 ; Automatic variable j
```

Static Model

(Output object of compiler …continued)

```
; line   7 :  k = i + j ;
      mov    a, e
      add    a, d                        ; Add i and j
      mov    !?L0003, a ; k              ; Substitute for K
; line   8 :  sub (i, j, k) ;

      mov    h, a                        ; Pass k through register H
      push   de
      pop    bc                          ; Passes j through register B
      mov    a, e                        ; Pass i through register A
      call   !_sub
; line   9 : }
      pop    de
      ret
; line   10 : void sub (char p1, char p2, char p3)
; line   11 : {
_sub:
      mov    l, a                        ; Allocate the 1st argument to l
      mov    a, h
      mov    _@KREG15, a                 ; Allocate the 3rd argument to shared area
; line   12 :  char a1, a2 ;
; line   13 :  a1 = 1<<p1 ;
      mov    a, l                        ; The 1st argument p1
      mov    c, a
      mov    a, #01H
      dec    c
      inc    c
      bz     $?L0006
      add    a, a
      dbnz   c, $$-2
?L0006:
      mov    _@KREG14, a    ; a1         ; Automatic variable a1 is in the shared area
; line   14 :  a2 = p2 + p3 ;
      mov    a, b                        ; The 2nd argument p2
      add    a,_@KREG15     ; p3         ; Adds the 3rd argument p3
      mov    _@KREG13, a    ; a2         ; Automatic variable a2 is in the shared area
; line   15 : }
      ret
```

Static Model

RESTRICTIONS

- Module of a static model cannot be link with a modules of a normal model. However, modules of a static model can be linked each other even if the maximum size of the shared area is different.
- Floating-point numbers are not supported. If the **float** and **double** keywords are described, a fatal error occurs.
- Arguments are limited to a maximum of 3 arguments and 6 bytes in total.
- It is impossible to use variable length arguments since arguments are not passed through stacks. Using variable length arguments causes an error.
- Arguments and return values of structures/unions cannot be used. The description of these arguments and values causes an error.
- The **noauto/norec/_ _leaf** functions cannot be used. A warning message is output and the descriptions are ignored (Refer to **11.5 (5) noauto functions**, **11.5 (6) norec functions**).
- Recursive functions cannot be used. As function arguments and the automatic variable area are statically secured, recursive functions cannot be used. An error is generated for recursive functions that can be detected by the compiler.
- A prototype declaration cannot be omitted. An error is generated if neither the no function's real definition nor a prototype declaration exist, in spite of there being a function call.
- Due to the restrictions of arguments and inability to use recursive functions, some standard libraries cannot be used.
- If the **-ZL** option has not been specified, a warning is output and processing is carried out as if the **-ZL** option was specified. **long** types are therefore always regarded as **int** types (see **11.5 (30) Type modification**).

COMPATIBILITY

<From another C compiler to this C compiler>

- When creating objects of normal model, source modification is not needed unless the **-SM** option is specified.
- To create a static model object, modifications are made according to the method above.

<From this C compiler to another C compiler>

- Source modification is not needed if re-compiling is performed by another compiler.

CAUTION

- Since arguments/automatic variables are secured statically, the contents of arguments/automatic variables in recursive functions may be destroyed. An error occurs when the function calls itself directly. However, no error occurs when the function calls itself after an other function is called since the compiler cannot detect this processing.
- During an interruption, the contents of arguments/automatic variables may be destroyed if the function being processed is called by interrupt servicing (interrupt functions and functions that are called by interrupt functions).
- During an interruption, save/return of the shared area is not executed even when the functions being processed are using the shared area.

(30) Type modification

Type Modification -ZI

(1) Change from int/short type to char type

FUNCTION

- **int** and **short** types are regarded as **char** type. In other words, **int** and **short** descriptions become equal to a **char** description.
- Details of the type modification are given as follows (Some **-QU** options are affected).

Table 11-14 Details of Type Modification (Change from int and short Type to char Type)

Type Described in C Source	Option	Type after Modification
short, **short int**, **int**	With **-QU**	**unsigned char**
short, **short int**, **int**	Without **-QU**	**signed char**
unsigned short, **unsigned short int**, **unsigned**, **unsigned int**	–	**unsigned char**
signed short, **signed short int**, **signed**, **signed int**	–	**signed char**

- Outputs warning message to the line where the **int** or **short** keywords first appeared in C source.
- The **-QC** option becomes effective regardless of whether it is specified. A warning message is output when there is no **-QC** option specification, and the **-QC** option becomes effective.
- If the **-ZA** option is specified at the same time (such as the **-ZAI** option), a warning message is output (only when **-W2** is specified).
- The following statement can be described by a type specifier and omitted, so are regarded as **char** type.
  - Arguments and returned values of functions
  - Type specifier omitted variables/function declaration
- The compiler executes the following macro definition automatically.

```
#define _ _FROM_INT_TO_CHAR_ _ 1
```

- Some standard libraries cannot be used.

USAGE

- The **-ZI** option is specified.

RESTRICTIONS

- **-ZI** specified and **-ZI** unspecified modules cannot be linked together.

Type Modification -ZL

(2) Change from long type to int type

FUNCTION

- **long** type is regarded as **int** type. In other words, a **long** description becomes equal to an **int** description.
- Details of the type modification are given as follows.

Table 11-15 Details of Type Modification (Change from long Type to int Type)

Type Described in C Source	Type after Modification
unsigned long, **unsigned long int**	**unsigned int**
long, **long int**, **signed long**, **signed long int**	**signed int**

- Outputs warning message to the line where the **long** keyword first appeared in C source.
- If the **-ZA** option is specified at the same time (**-ZAL**), a warning message is output (only when **-W2** is specified).
- The compiler executes the following macro definition automatically.

```
#define _ _FROM_LONG_TO_INT_ _ 1
```

- Some standard libraries cannot be used.

USAGE

- The **-ZL** option is specified.

RESTRICTIONS

- **-ZL** specified and **-ZL** unspecified modules cannot be linked together.

(31) Changing function call interface

Changing Function Call Interface -ZO

FUNCTION

- Arguments are passed by the former function interface specification (the CC78K0 Ver.2.11 compatible, only stack is used). For the details of function interface, refer to **11.7 Function Call Interface**.

USAGE

- The **-ZO** option is specified during compile option.

RESTRICTION

- The modules to which the **-ZO** option is specified and the modules to which the **-ZO** option is not specified cannot be linked to one another.

(32) Pascal function

Pascal Function _ _pascal

FUNCTION

- Generates the code that the correction of the stack used for the place of arguments during the function call is performed on the called function side, not on the side calling the function.

EFFECT

- Object code can be shortened if a lot of function call appears.

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 pascal function, the keywords **norec/_ _interrupt/_ _interrupt_brk/_ _rtos_interrupt/_ _flash/_ _flashf/ _ _banked 1** to **15** cannot be specified. If they are specified, in the case of the norec keyword, the **_ _pascal** keyword is disregarded and in the case of the **_ _interrupt/_ _interrupt_brk/_ _rtos_interrupt/_ _flash/ _ _flashf/_ _banked 1** to **15** 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** keyword appears and **_ _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.
- Pascal functions are not supported when the static model specification option (**-SM**) is specified. If **-SM** is specified when using the pascal function, a warning message is output to the place where the **_ _ pascal** keyword first appeared, and the **_ _ pascal** keyword in the input file is ignored.

EXPLANATION

- The **-ZR** option enables the change of all functions to the pascal function. However, if the pascal function is used for the functions that have few calls, the object code may increase.

Pascal Function _ _pascal

EXAMPLE

(C source)

```
_ _pascal int func(int a, int b, int c);
void main()
{
      int ret_val;

      ret_val = func(5, 10, 15);
      }
      _ _pascal int func(int a, int b, int c);
      {
            return (a + b + c);
      }
```

(Output object of compiler)

```
_main :
              push   hl
              movw   ax, #0FH       ; A 4-byte stack is consumed by arguments
              push   ax             ;
              mov    x, #0AH        ;
              push   ax             ;
              mov    x, #05H        ;
              call   !_func
                                    ; The stack is not modified here.
              movw   ax, bc
              movw   hl, ax
              pop    hl
              ret

_func :
              push   hl
              push   ax
              movw   ax, sp
              movw   hl, ax
              mov    a, [hl + 6]
              add    a, [hl]
```

Pascal Function __pascal

(Output object of compiler …continued)

```
        xch     a, x
        mov     a, [hl + 7]
        addc    a, [hl + 1]
        xch     a, x
        add     a, [hl + 8]
        xch     a, x
        addc    a, [hl + 9]
        movw    bc, ax
        pop     ax
        pop     hl
        pop     de              ; Obtains the return address
        pop     ax              ;
        pop     ax              ; The 4-byte stack that was consumed by the caller is
                                ; modified.
        push    de              ; Return address is reloaded
        ret
```

COMPATIBILITY

<From other C compiler to this C compiler>

- If the reserved word, **_ _ pascal** is not used, modification is not required.
- To change to the Pascal function, change according to the above method.

<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.

(33) Automatic pascal functionization of function call interface

Automatic Pascal Functionization of Function Call Interface -ZR

FUNCTION

- With the exception of **norec/_ _interrupt/_ _interrupt_brk/_ _rtos_interrupt/_ _flash/_ _flashf/_ _banked 1** to **15/**variable length argument functions, **_ _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.
- It is impossible to specify the static model specification option (**-SM**) and the **-ZR** option at the same time. If specified, a warning message is output and the **-ZR** option is ignored.
- Since the mathematical function standard library does not support the pascal function, the **-ZR** option cannot be used when the mathematical function standard library is used.

Remark For pascal function call interface, refer to **11.7.6 Pascal function call interface**.

(34) Flash area allocation method

Flash Area Allocation Method -ZF

Caution Do not use this flash function for the devices that have no flash area self-rewriting function. Operation is not guaranteed if it is used.
This function enables the function of rewriting the flash memory of devices.

FUNCTIONS

- Generates an object file located in the flash area.
- External variables in the flash area cannot be referred to from the boot area.
- External variables in the boot area can be referred to from the flash area.
- The same external variables and the same global functions cannot be defined in a boot area program and a flash area program.

EFFECT

- Enables locating a program in the flash area.
- Enables using function linking with a boot area object created without specifying the **-ZF** option.

USAGE

- Specify the **-ZF** option during compilation.

RESTRICTION

- Use start-up routines or library for the flash area.

(35) Flash area branch table

Flash Area Branch Table #pragma ext_table

Caution Do not use this flash function for the devices that have no flash area self-rewriting function. Operation is not guaranteed if it is used.
This function enables the function of rewriting the flash memory of devices.

FUNCTIONS

- Determines the first address of the branch table for the start-up 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 3 bytes of area. 32 from the first address of the branch table is reserved as dedicated interrupt functions. Ordinary functions are located after the "first address of branch table +3 * 32."
- The branch table occupies 3* (32 + **ext_func** ID max. value + 1) bytes of area. For the **ext_func** ID value, refer to **11.5 (36) Function of function call from boot area to flash area**.

EFFECT

- A start-up routine and interrupt function can be located in the flash area.
- A function calls can be performed from the boot area to the flash area.

USAGE

- The following **#pragma** instruction specifies the first address of the flash area branch table.

```
#pragma ext_table branch-table-first-address
```

Describe the **#pragma** instruction at the beginning of C source.

- The following items can be described before the **#pragma** instruction:
  - Comments
  - **#pragma** instructions other than **#pragma ext_func, #pragma vect** with **-ZF** specification, **#pragma interrupt**, or **#pragma rtos_interrupt**.
  - Instructions not to generate the definition/reference of variables or functions among the preprocess instructions.

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 80H to 0FF80H. However, match the first address value with the flash start address which is specified in 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 (**_@vect00** to **_@vect3e**) in accordance with the specified first address of the branch table. The default is 2000H in the interrupt vector library. To specify the value other than 2000H, reconfigure the library as shown below.

1. Change the place of H in ITBLTOP EQU 2000H of **vect.inc** in the /NECTools32/SRC/CC78K0/SRC directory to the specified address.
2. Run /NECTools32/SRC/CC78K0/BAT/repvect.bat in DOS prompt, and update library by assembly. Copy the updated library /NECTools32/SRC/CC78K0/LIB to /NECTools32/LIB78K0 to be used for link.

Caution The above directory may differ depending on the installation method.

COMPATIBILITY

<From another C compiler to this C compiler>

- If **#pragma ext_table** is not used, correction is not necessary.
- To specify the first address of the flash area branch table, change the address in accordance with **USAGE** above.

<From this C compiler to another C compiler>

- Delete the **#pragma ext_table** instruction or divide it by **#ifdef**.
- To specify the first address of the flash area branch table, the following change is required.

EXAMPLE

(C source)

To generate a branch table after the address 2000H and place the interrupt function:

```
#pragma ext_table 0x2000
#pragma interrupt INTP0 intp

void intp ()
{
}
```

Flash Area Branch Table #pragma ext_table

(Output code)

(a) To place the interrupt function to the boot area (no **-ZF** specified)

```
         PUBLIC  _intp
         PUBLIC  _@vect06
@@CODE   CSEG
_intp :
         reti

@@VECT06 CSEG    AT      0006H
_@vect06 :
         DW      _intp
```

- Sets the first address of the interrupt function in the interrupt vector table.

(b) To place the interrupt function in the flash area (**-ZF** specified)

```
         PUBLIC  _intp
@ECODE   CSEG
_intp :
         reti

@EVECT06        CSEG    AT   02009H
         br      !_intp
```

- Sets the first address of the interrupt function in the branch table.
- The address value of the branch table is 2000H + 3* (0006H/2) since the first address of the branch table is 2000H and the interrupt vector address (2 bytes) is 0006H.
- The interrupt vector library performs the setting of the address 2009H in the interrupt vector table.

(Library for interrupt vector 06)

```
         PUBLIC  _@vect06

@@VECT06        CSEG    AT   0006H
_@vect06 :
         DW      2009H
```

(36) Function of function call from boot area to flash area

Function of Function Call from Boot Area to Flash Area #pragma ext_func

Caution Do not use this flash function for the devices that have no flash area self-rewriting function. Operation is not guaranteed if it is used.
This function enables the function of rewriting the flash memory of devices.

FUNCTIONS

- Function calls from the boot area to the flash area are executed via the flash area branch table.
- From the flash area, functions in the boot area can be called directly.

EFFECT

- It becomes possible to call a function in the flash area from the boot area.

USAGE

- The following **#pragma** instruction specifies the function name and ID value in the flash area called from the boot area.

```
#pragma ext_func function-name ID value
```

This **#pragma** instruction is described at the beginning of the C source. The following items can be described before this **#pragma** instruction.

- Comments
- Instructions not to generate the definition/reference of variables or functions among the preprocess instructions.

RESTRICTIONS

- The ID value is set at 0 to 255 (0xFF).
- **#pragma ext_table** does not exist before **#pragma ext_func**, it results in an error.
- For the same function with a different ID value and a different function with the same ID value, an error 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 of Function Call from Boot Area to Flash Area #pragma ext_func

COMPATIBILITY

<From another C compiler to this C compiler>

- If the **#pragma ext_func** is not used, no corrections are necessary.
- To perform the function call from the boot area to the flash area, make the change in accordance with **USAGE** above.

<From this C compiler to another C compiler>

- Delete the **#pragma ext_func** instruction or divide it by **#ifdef**.
- To perform the function call from the boot area to the flash area, the following change is required.

EXAMPLE

In the case that the branch table is generated after address 2000H and functions f1 and f2 in the flash area are called from the boot area.

(C source)

(1) Boot area side

```
#pragma ext_table 0x2000
#pragma ext_func f1 3
#pragma ext_func f2 4

extern void f1(void);
extern void f2(void);

void func()
{
  f1();
  f2();
}
```

Function of Function Call from Boot Area to Flash Area #pragma ext_func

(C source …continued)

(2) Flash area side

```
#pragma ext_table 0x2000
#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 2000H + 3*32 + 3*3.
- #pragma ext_func f2 4 means that the branch destination to function f2 is located in branch table address 2000H + 3*32 + 3*4.
- 3*32 bytes from the beginning of the branch table is exclusively for interrupt functions (including the start-up routine).

(Output object of compiler)

(1) Boot area side (without -ZF specification)

```
@@CODE   CSEG
_func:
         call    !02069H
         call    !0206CH
         ret
```

(2) Flash area side (with -ZF specification)

```
@ECODE   CSEG
_f1:
         ret
_f2:
         ret

@EXT03   CSEG    AT      02069H
         br      !_f1
         br      !_f2
```

(37) Firmware ROM function

Firmware ROM Function _ _flash

Caution Do not use this flash function for the devices that have no flash area self-rewriting function. Operation is not guaranteed if it is used.
This function enables the function of rewriting the flash memory of devices.

FUNCTIONS

- This calls a firmware ROM function which self-writes to the flash memory via the interface library positioned between the firmware ROM function and the C language function.
- In the interface library call interface, the first argument is passed to the register and the second and subsequent arguments are transferred to the stack. The first argument's register is as follows.

1, 2-byte data	**AX**
4-byte data	**AX** (low-order), **BC** (high-order)

- It is necessary that the interface library be set to the return the values in the following registers according to the size of return values.

1- or 2-byte data	**BC**
Pointer	**BC**
4-byte data	**BC** (low-order), **DE** (high-order)

EFFECT

- The operations related to the firmware ROM function can be described at the C source level.

USAGE

- During interface library prototype declaration, **_ _flash** attributes are added to the top.

RESTRICTIONS

- Function calls by a function pointer are not supported.
- When a function with **_ _flash** is defined, it results in an error.
- When the static model is specified, 4-byte data are not supported.

COMPATIBILITY

<From another C compiler to this C compiler>

- If the reserved word **_ _flash** is not used, corrections are not necessary.
- If you desire to change the firmware ROM function, use the **USAGE** above.

<From this C compiler to another C compiler>

- Possible using **#define** (refer to C source corrections).
- 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.

(38) 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    a, !_e
     mov    x, a                      ; Do not execute int expansion
     push   ax
     mov    a, !_d
     mov    x, a                      ; Do not execute int expansion
     call   !_func1
     pop    ax
     mov    a, c
     mov    !_c, a
; line  6   c = func2 (d, e) ;
     mov    a, !_e
     mov    x, #00H ; 0
     xch    a, x                      ; Execute int expansion since there is no prototype declaration
     push   ax
     mov    a, !_d
     mov    x, #00H ; 0
     xch    a, x                      ; Execute int expansion since there is no prototype declaration
     call   !_func2
     pop    ax
     mov    a, c
     mov    !_c, a
     ret
; line  8:  unsigned char func1 (unsigned char x, unsigned char y)
_func1 :
     push   hl
     push   ax
     movw   ax, sp
     movw   hl, ax
     mov    a, [hl]
     xch    a, x
     mov    a, [hl + 6]
     movw   hl, ax
; line 10:  return x + y ;
     mov    a, l
     add    a, h
     mov    c, a                      ; Do not execute int expansion
     pop    ax
     pop    hl
     ret
```

Method of int Expansion Limitation of Argument/Return Value -ZB

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 needed.

(39) Array offset calculation simplification method

Array Offset Calculation Simplification Method -QW2, -QW3

FUNCTION

- When calculating the offset of **char/unsigned char/unsigned int/short/unsigned short** types and the index is an **unsigned char**-type variable, a code to calculate only low-order bytes is generated based on the presumption that there is no carry-over.
- When the **-QW2** option is specified, a code to calculate only low-order bytes for the offset is generated only when referencing the sequence of the saddr area configuration with an unsigned char variable.
- When the **-QW3** option is specified, the code to calculate only low-order bytes for the offset is generated when referencing the sequence with an unsigned char variable regardless of the configured area.

EFFECT

- Realizes object code reduction and execution speed improvement since the offset calculation code is simplified.

USAGE

- Specifies the **-QW2** and **-QW3** options during compilation.

EXAMPLE

(C source)

```
unsigned char c ;
unsigned char ary [10] ;
sreg unsigned char sary [10] ;
void main ()
{
        unsigned char a ;

        a = ary [c] ;
        a = sary [c] ;
}
```

Array Offset Calculation Simplification Method -QW2, -QW3

(Output of compiler object)

When **-QW3** is specified

```
_main :
        push    hl
        push    ax
        movw    ax, sp
        movw    hl, ax
; line     6 : unsigned char a ;
; line     7 :
; line     8 : a = ary [c] ;
        mov     a, !_c
        add     a, #low (_ary)
        mov     e, a                    ; Calculate only low-order bytes
        mov     d, #high (_ary)
        mov     a, [de]
        mov     [hl + 1], a             ; a
; line      9 :a = sary [c] ;
        mov     a, !_c
        add     a, #low (_sary)
        mov     e, a                    ; Calculate only low-order bytes
        mov     d, #0FEH ; 254
        mov     a, [de]
        mov     [hl + 1], a             ; a
; line    10 :}
        pop     ax
        pop     hl
        ret
```

RESTRICTIONS

- If the configuration addresses of sequence that is the target for offset calculation simplification is over the border of 256 bytes, the program may operate incorrectly.
- **-QW4** and **-QW5** are not supported.

COMPATIBILITY

<From another C compiler to this C compiler>

- Assign the layout so that it does not exceed 256 bytes. Alternatively, do not specify the **-QW2** and **-QW3** options.

<From this C compiler to another C compiler>

- No modification is needed.

(40) Register direct reference function

Register Direct Reference Function #pragma realregister

FUNCTION

- Output the code that accesses the object register with direct in-line expansion instead of function call, and generates an object file.
- When there is no **#pragma** directive, the register direct reference function is regarded as an ordinary function.

EFFECT

- Due to the C description, register access can be performed easily.

USAGE

- This function is described in the same format as a function call (Refer to **Register direct reference function list** later in this chapter).
  There are 21 types of register direct reference function names.

```
_ _geta, _ _seta, _ _getax, _ _setax, _ _getcy, _ _setcy, _ _set1cy, _ _clr1cy
_ _not1cy, _ _inca, _ _deca, _ _rora, _ _rorca, _ _rola, _ _rolca, _ _shla
_ _shra, _ _ashra, _ _nega, _ _coma, _ _absa
```

- By using the **#pragma realregister** directive in a module, use of register direct reference function is declared.
  The followings can be described before the **#pragma realregister** directive.
  - Comments
  - Other **#pragma** directives
  - Preprocess directives that do not generate variable definitions/references nor function definitions/references

EXAMPLE

(C source)

```
#pragma realregister
unsigned char c = 0x88,  d, e ;
void main ()
{
     _ _seta (c) ;              /* Sets the  variable of C in A register         */
     _ _shla () ;               /* Logically shifts 1 bit to left                */
     d = _ _geta () ;           /* Sets the value of A register in variable d    */
     if (_ _getcy () ) {        /* Refers CY (checks overflow)                   */
          e = 1 ;               /* Sets e to 1 when CY = = 1                     */
     }
}
```

Register Direct Reference Function #pragma realregister

(Output object of compiler)

```
_main :
; line    5 : _ _seta (c) ;        /* Sets the variable of C in A register     */
        mov    a, !_c
; line    6 : _ _shla () ;         /* Logically shift 1 bit to left            */
        add    a, a
; line    7 : d = _ _geta () ;     /* Sets value of A register in variable d   */
        mov    !_d, a
; line    8 : if (_ _getcy () ) {/* Refers CY (checks overflow)               */
        bnc    $?L0003
; line    9             e = 1 ;  /* Sets e to 1 when CY = = 1                  */
        mov    a, #01H         ; 1
        mov    !_e, a
?L0003 :
; line    10 :}
; line    11 :}
        ret
```

[Register direct reference function list]

(1) **`unsigned char _ _geta (void) ;`**
Obtains the value of the A register.

(2) **`void _ _seta (unsigned char x) ;`**
Sets x in the A register.

(3) **`unsigned int _ _getax (void) ;`**
Obtains the value of the AX register.

(4) **`void _ _setax (unsigned int x) ;`**
Sets x in the AX register.

(5) **`bit _ _getcy (void) ;`**
Obtains the value of the **CY** flag.

(6) **`void _ _setcy (unsigned char x) ;`**
Sets the lower 1 bit of x in the **CY** flag.

(7) **`void _ _set1cy (void) ;`**
Generates the set1 **CY** instruction.

Register Direct Reference Function #pragma realregister

(8) `void _ _clr1cy (void) ;`
Generates the clr1 **CY** instruction.

(9) `void _ _not1cy (void) ;`
Generates the not1 **CY** instruction.

(10) `void _ _inca (void) ;`
Generates the inc **a** instruction.

(11) `void _ _deca (void) ;`
Generates the dec **a** instruction.

(12) `void _ _rora (void) ;`
Generates **1** ror **a**, instruction.

(13) `void _ _rorca (void) ;`
Generates **1** rorc **a**, instruction.

(14) `void _ _rola (void) ;`
Generates **1** rol **a**, instruction.

(15) `void _ _rolca (void) ;`
Generates **1** rolc **a**, instruction.

(16) `void _ _shla (void) ;`
Generates the code that performs logical-shift of the A register 1 bit to the left.

(17) `void _ _shra (void) ;`
Generates the code that performs a logical-shift of the A register 1 bit to the right.

(18) `void _ _ashra (void) ;`
Generates the code that performs an arithmetic-shift of the A register 1 bit to the right.

(19) `void _ _nega (void) ;`
Generates the code that obtains 2's complement in the A register.

(20) `void _ _coma (void) ;`
Generates the code that obtains 1's complement in the A register.

(21) `void _ _absa (void) ;`
Generates the code that obtains the absolute value of the A register.

Register Direct Reference Function #pragma realregister

RESTRICTIONS

- The function name for that register direct reference cannot be not used as function name. The register direct reference function is described in lowercase letters. A function described in uppercase letters are regarded as an ordinary function.
- The values of the **A** and **AX** registers, and the **CY** flag that are set by the **_ _ seta**, **_ _setax**, and **_ _setcy** functions are not retained in the next code generation.
- The timing that is referenced by a and **AX** registers, and the **CY** flag with the **_ _ geta**, **_ _getax**, and **_ _getcy** function are corresponds to the evaluation sequence of the expression.

COMPATIBILITY

<From another C compiler to this C compiler>

- If the register direct reference function is not used, modification is not necessary.
- To change to the register direct referencing function, use the method above.

<From this C compiler to another C compiler>

- The **"#pragma realregister**" directive should be deleted or delimited using **#ifdef**. Register direct reference function names can be used as function names.
- When using "**pragma realregister**" as a register direct reference function, the change to the source program must conform to the specification of the C compiler (**#asm**, **#endasm**, or **asm();**, etc.).

CAUTION

- There is no guarantee that **CY**, **A**, **AX** will be saved as intended before the register direct reference function is executed. Accordingly, it is recommended to use this function before values change by describing it in the first term of the expansion.

(41) [HL + B] based indexed addressing utilization method

Based Indexed Addressing Utilization Method -QE

FUNCTION

- When the index is the unsigned char variable while referring the **char/unsigned char**-type arrangement and **char/unsigned char**-type pointer, codes that include [HL + B] based indexed addressing is generated.

EFFECT

- The object code is reduced and the execution speed improved.

USAGE

- The **-QE** option is specified during compilation.

EXAMPLE

(C source)

```
unsigned char c, d ;
unsigned char ary [10] ;
char *p ;
void main ()
{
        ary [c]  *= d + 1 ;

        * (p + c) *= 4 ;
}
```

Based Indexed Addressing Utilization Method -QE

(Output object of compiler)

When **-SM** and **-QCE** are specified

```
_main :
; line   6 : ary [c] *= d + 1 ;
     mov    a, !_d
     inc    a
     mov    x, a
     mov    a, !_c
     mov    b, a
     movw   hl, #_ary
     mov    a, [hl + b]             ; Uses [HL + B] based indexed addressing
     mulu   x
     mov    a, x
     mov    [hl + b], a             ; Uses [HL + B] based indexed addressing
; line   7 :
; line   8 : * (p + c) *= 4 ;
     mov    a, !_c
     mov    b, a
     movw   ax, !_p
     movw   hl, ax
     mov    a, [hl + b]             ; Uses [HL + B] based indexed addressing
     add    a, a
     add    a, a
     mov    [hl + b], a             ; Uses [HL + B] based indexed addressing
; line   9 : }
     ret
```

RESTRICTIONS

- The object code may increase some source description. In the normal model, this function is disabled.

COMPATIBILITY

<From another C compiler to this C compiler>

- Modification is not necessary.

<From this C compiler to another C compiler>

- Modification is not necessary.

(42) On-chip firmware self-programming subroutine direct call function

On-Chip Firmware Self-Programming Subroutine Direct Call Function #pragma hromcall

Caution Do not use this flash function for the devices that have no flash area self-rewriting function. Operation is not guaranteed if it is used.
This function enables the function of rewriting flash memory of devices.

FUNCTION

- An object file is generated by the output of the on-chip firmware self-programming subroutine direct call code to an object with direct inline expansion instead of function call.
- When there is no **#pragma** directive, the on-chip firmware self-programming subroutine direct call function is regarded as an ordinary function.
- The **_ _setup** function sets **SP** (stack pointer) to the specified address.
- The **_ _hromcall** function calls the specified address by switching the register bank to bank 3 temporarily and setting the function number in the C register and the entry RAM area beginning address in HL, respectively. The values in the B register are the return values.
- The **_ _hromcalla** function calls the specified address by switching the register bank to bank 3 temporarily and setting the function number in the C register and the entry RAM area beginning address in HL. The values in B register are a return value.
- The following functions call the specified address by switching the register bank to bank 3 temporarily and set the entry RAM area beginning address in HL.

```
_ _FlashEnv, _ _FlashSetEnv, _ _FlashGetInfo, _ _FlashAreaBlankCheck
_ _FlashAreaPreWrite, _ _FlashAreaErase, _ _FlashAreaWriteBack
_ _FlashByteWrite, _ _FlashWordWrite, _ _FlashAreaIVerify, _ _FlashByteRead
_ _FlashBlockBlankCheck, _ _FlashBlockPreWrite, _ _FlashBlockErase
_ _FlashBlockWriteBack, _ _FlashBlockIVerify
```

EFFECT

- Due to the C description, calling the on-chip firmware self-programming subroutine can be performed easily.

On-Chip Firmware Self-Programming Subroutine Direct Call Function #pragma hromcall

USAGE

- The function is described in the same format as a function call. The following 19 functions are on-chip firmware self-programming subroutine direct call function names (Refer to **On-chip firmware self-programming subroutine direct call function list**, described later in this chapter).

```
_ _hromcall, _ _hromcalla, _ _setsp, _ _FlashEnv, _ _FlashSetEnv
_ _FlashGetInfo, _ _FlashAreaBlankCheck, _ _FlashAreaPreWrite
_ _FlashAreaErase, _ _FlashAreaWriteBack, _ _FlashByteWrite
_ _FlashWordWrite, _ _FlashAreaIVerify, _ _FlashByteRead
_ _FlashBlockBlankCheck, _ _FlashBlockPreWrite, _ _FlashBlockErase
_ _FlashBlockWriteBack, _ _FlashBlockIVerify
```

- The **pragma hromcall** directive in a module performs declaration of the use of on-chip firmware self-programming subroutine direct call. However, the following items can be described before **#pragma hromcall**.
  - Comments
  - Other **#pragma** directives
  - Preprocess directives that do not generate variable definitions/references or function definitions/references

EXAMPLE

(C source)

```
#pragma di
#pragma sfr
#pragma hromcall
unsigned char entryram [32] ;
unsigned char ret ;
void func ()
{
        /* Interrupt disabled */
        DI () ;
        /* Enter self-programming mode */
        FLSPM0 = 1 ;

        /* Call FlashEnv subroutine call */
        _ _hromcall (0x8100, 0, entryram) ;
```

On-Chip Firmware Self-Programming Subroutine Direct Call Function — #pragma hromcall

(C source …continued)

```
        /* Set write time data */
        entryram [7]  = 0x20 ;
        /* Set delete time data */
        entryram [8]  = 0x4c ;
        entryram [9]  = 0x4c ;
        entryram [10]  = 0x00 ;
        /* Set convergence time data */
        entryram [11]  = 0x01 ;
        entryram [12]  = 0x3d ;
        /* Call FlashSetEnv subroutine */
        ret = _ _FlashSetEnv (entryram) ;
             .
             .
             .
        }
```

(Output object of compiler)

```
    _func ;
          di
    ; line   8  : /* Interrupt disabled */
    ; line   9  : DI () ;
    ; line   10 : /* Enter self-programming mode */
    ; line   11 : FLSPM0 = 1 ;
          set1    FLSPM0
    ; line   12 :
    ; line   13 : /* Call FlashEnv subroutine call */
    ; line   14 : _ _hromcall (0x8100, 0, entryram) ;
          push    psw                                 ; Save current register bank
          sel     rb3                                 ; Switch to bank 3
          movw    hl, #_entryram
          mov     c, #00H        ; 0
          call    !08100H
          pop     psw                                 ; Return to current register bank
          mov     a, 0FEE3H
    ; line   15 : /* Set write time data*/
    ; line   16 : entryram [7] = 0x20 ;
          mov     a, #020H ; 32
          mov     !_entryram + 7, a
```

On-Chip Firmware Self-Programming Subroutine Direct Call Function #pragma hromcall

(Output object of compiler …continued)

```
    ; line    17 : /* Set delete time data*/
    ; line    18 : entryram [8] = 0x4c ;
          mov      a, #04CH ; 76
          mov      !_entryram + 8, a
    ; line    19 : entryram [9] = 0x4c ;
          mov      !_entryram + 9, a
    ; line    20 : entryram [10] = 0x00 ;
          mov      a, #00H       ; 0
          mov      !_entryram + 10, a
    ; line    21 : /* Set convergence time data*/
    ; line    22 : entryam [11] = 0x01 ;
          inc      a
          mov      !_entryram + 11, a
    ; line    23 : entryram [12] = 0x3d ;
          mov      a, #03DH ; 61
          mov      !_entryram + 12, a
    ; line    24 : /* Calls FlashSetEnv subroutine*/
    ; line    25 : ret = _ _FlashSetEnv (entryram) ;
          push     psw                              ; Save current register bank
          sel      rb3                              ; Switch to bank 3
          movw     hl, #_entryram
          mov      c, #01H       ; 1
          call     !08100H
          pop      psw                              ; Return to current bank register
          mov      a, 0FEE3H
          mov      !_ret, a
            .
            .
            .
          ret
```

On-Chip Firmware Self-Programming Subroutine Direct Call Function #pragma hromcall

[On-chip firmware self-programming subroutine direct call function list]

(1) unsigned char _ _hromcall (unsigned int entryaddr, unsigned char funcno, void *entrydata) ;

Calls the entryaddr address after switching to register bank 3 temporarily and setting **entrydata** in the **HL** register and **funcno** in the **C** register, respectively. The value in the **B** register is a return value.

(2) unsigned char _ _hromcalla (unsigned int entryaddr, unsigned char funcno, void *entrydata) ;

Calls the entryaddr address after switching to register bank 3 temporarily and setting **entrydata** in the **HL** register and **funcno** in the **C** register, respectively. The value in the **A** register is the return value.

(3) void _ _setsp (unsigned int spaddr) ;

Sets the value of **spaddr** in **SP** (stack pointer).

(4) void _ _FlashEnv (void *entryaddr) ;

Calls the initialization subroutine of the flash subroutine after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is the return value.

(5) unsigned char _ _FlashSetEnv (void *entryaddr) ;

Calls the parameter setting subroutine after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is the return value.

(6) unsigned char _ _FlashGetInfo (void *entryaddr) ;

Calls the flash information read subroutine after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the B register is the return value.

(7) unsigned char _ _FlashAreaBlankCheck (void *entryaddr) ;

Calls the blank check subroutine in the specified area after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is the return value.

(8) unsigned char _ _FlashAreaPreWrite (void *entryaddr) ;

Calls the pre-write subroutine in the specified area after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is set the return value.

(9) unsigned char _ _FlashAreaErase (void *entryaddr) ;

Calls the delete subroutine in the specified area after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is set the return value.

(10) unsigned char _ _FlashAreaWriteBack (void *entryaddr) ;

Calls the write-back subroutine in the specified area after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is the return value.

On-Chip Firmware Self-Programming Subroutine Direct Call Function #pragma hromcall

(11) unsigned char _ _FlashByteWrite (void *entryaddr) ;
Calls the successive write subroutine in byte units after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is the return value.

(12) unsigned char _ _FlashWordWrite (void *entryaddr) ;
Calls the successive write subroutine in word units after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is set the return value.

(13) unsigned char _ _FlashArealVerify (void *entryaddr) ;
Calls the internal verify subroutine in the specified area after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the B register is set the return value.

(14) unsigned char _ _FlashByteRead (void *entryaddr) ;
Calls the read subroutine in byte units after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **A** register is set the return value.

(15) unsigned char _ _FlashBlockBlankCheck (void *entryaddr) ;
Calls the blank check subroutine in the specified block after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is the return value.

(16) unsigned char _ _FlashBlockPreWrite (void *entryaddr) ;
Calls the pre-write subroutine in the specified area after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is set to be a return value.

(17) unsigned char _ _FlashBlockErase (void *entryaddr) ;
Call the delete subroutine in the specified block after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is the return value.

(18) unsigned char _ _FlashBlockWriteBack (void *entryaddr) ;
Calls the write-back subroutine in the specified block after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is set the return value.

(19) unsigned char _ _FlashBlockIVerify (void *entryaddr) ;
Call the internal verify subroutine in the specified block after switching to register bank 3 temporarily and setting **entryaddr** in the **HL** register. The value in the **B** register is set to be a return value.

On-Chip Firmware Self-Programming Subroutine Direct Call Function #pragma hromcall

RESTRICTIONS

- Function names for on-chip firmware self-programming subroutine direct call cannot be used for function name.
- This function is not available in devices that do not incorporate the firmware in which the self-programming subroutine direct call is written.
- The subroutine that is called by the **_ _Flash**~ function may not be supported in some devices. Functions that are not supported cannot be used.
- If the specifications of the on-chip firmware self-programming subroutine are not as follows, this function cannot be used.
  - Uses register bank 3
  - Sets function number in the C register
  - Sets the beginning address of the entry ARM area in the HL register
- Only a constant can be specified for the first and second arguments in the **_ _hromcall**, **_ _hromcalla** functions. Specifications other than a constant result in an error.
- Functions called by the **_ _Flash~** function can be used if the specifications for the on-chip firmware subroutine do not correspond with the following table, **Table 11-16**.

Table 11-16 _ _Flash~ Function Name – Subroutine Name, Firmware Entry Address and Function Number Corresponding Table

Function Name	Subroutine Name	Firmware ROM Entry Address	Function Number
_ _FlashEnv	_FlashEnv	8100H	00H
_ _FlashSetEnv	_FlashSetEnv	8100H	01H
_ _FlashGetInfo	_FlashGetInfo	8100H	02H
_ _FlashAreaBlankCheck	_FlashAreaBlankCheck	8100H	10H
_ _FlashAreaPreWrite	_FlashAreaPreWrite	8100H	20H
_ _FlashAreaErase	_FlashAreaErase	8100H	30H
_ _FlashAreaWriteBack	_FlashAreaWriteBack	8100H	40H
_ _FlashByteWrite	_FlashByteWrite	8100H	50H
_ _FlashWordWrite	_FlashWordWrite	8100H	51H
_ _FlashAreaIVerify	_FlashAreaIVerify	8100H	60H
_ _FlashByteRead	_FlashByteRead	8100H	70H
_ _FlashBlockBlankCheck	_FlashBlockBlankCheck	8100H	11H
_ _FlashBlockPreWrite	_FlashBlockPreWrite	8100H	21H
_ _FlashBlockErase	_FlashBlockErase	8100H	31H
_ _FlashBlockWriteBack	_FlashBlockWriteBack	8100H	41H
_ _FlashBlockIVerify	_FlashBlockIVerify	8100H	61H

On-Chip Firmware Self-Programming Subroutine Direct Call Function — #pragma hromcall

COMPATIBILITY

<From another C compiler to this C compiler>

- Modification is not needed if the on-chip firmware self-programming subroutine direct call function is not used.
- When changing to the on-chip firmware self-programming subroutine direct call function, use the method above.

<From this C compiler to another C compiler>

- The **"#pragma hromcall**" statement should be deleted or delimited using **#ifdef**. The function name for on-chip firmware self-programming subroutine direct call can be used as a function name.
- To use as "**pragma hromcall**" as the function for on-chip firmware self-programming subroutine direct call, changes to the source program must conform to the specification of each C compiler (**#asm**, **#endasm**, or **asm();**, etc).

CAUTION

- Before calling this function, arguments should be set in the RAM area (Refer to the user's manual of the relevant device for the values set in the entry RAM area).
- This function does not perform either interrupt disable processing or transition to self-programming mode processing. Accordingly, these processes should be performed before using this function.
- For the firmware entry address that is set in the **_ _hromcall** and **_ _hromcall** function and values that are set in the function number, refer to the user's manual of the relevant device.

(43) _ _ flashf function

_ _ flashf Function _ _ flashf

Caution Do not use this flash function for the devices that have no flash area self-rewriting function. Operation is not guaranteed if it is used.
This function enables the function of rewriting the flash memory of devices.

FUNCTION

- After storing program status word in the stack at the beginning of a function, this function switches to interrupt disable and register bank 3.
- A program status word that is stored in the stack is restored at the end of a function.
- The function for **11.5 (42) On-chip firmware self-programming direct subroutine call function** becomes valid regardless of whether or not the #pragma hromcall declaration exist.
- The function caller calls by setting arguments to **A** (1-byte data) or **AX** (2-byte data); the function definition side copies the arguments that are passed into A or AX to the saddr area ([FEBAH to FEBFH] in normal mode).
- Automatic variables are allocated to the saddr area ([FEBAH to FEBFH] in normal mode) including register variables.

EFFECT

- When **11.5 (42) On-chip firmware self-programming direct subroutine call function** is written in a function in which the **_ _flashf** attributes are added, a code, which switches to bank save/restore and register bank 3 at each call, is not generated.

USAGE

- During a function declaration **_ _flashf** attributes are added to the beginning.

_ _ flashf Function _ _ flashf

EXAMPLE

(C source)

```
#pragma di
#pragma sfr
#pragma hromcall
unsigned char entryram [32] ;
unsigned char ret ;
_ _flashf void func ()
{
        /* Move to self-programming mode */
        FLSPM0 = 1 ;
        /* Call _FlashEnv subroutine */
        _ _hromcall (0x8100, 0, entryram);
        /* Set write time data */
        entryram [7] = 0x20 ;
        /* Set delete time data */
        entryram [8] = 0x4c ;
        entryram [9] = 0x4c ;
        entryram [10] = 0x00 ;
        /* Set convergence time */
        entryarm [11] = 0x01 ;
        entryram [12] = 0x3d ;
        /* Call _FlashSetEnv subroutine */
        ret = _ _FlashSetEnv (entryram) ;
          .
          .
          .
}
```

_ _ flashf Function   _ _ flashf

(Output object of compiler)

```
_func ;
     push   psw       ; Save current register bank              ; Compiler generates these 3
     di               ; Interrupt disabled                      ; lines automatically
     sel    rb3       ; Switch to bank 3                        ;
; line 7 :  /* Move to self-programming mode */
; line 8 :  FLSPM0 = 1 ;
     set1   FLSPM0
; line 9 :
; line 10 : /* Call _FlashEnv subroutine */
; line 11 : _ _hromcall (0x8100, 0, entryram) ;
     movw   hl, #_entryram
     mov    c, #00H          ; 0
     call   !08100H
; line 12 : /* Set write time data */
; line 13 : entryram [7] = 0x20 ;
     mov    a, #020H ; 32
     mov    [hl + 7], a
; line 14 : /* Set delete time data */
; line 15 : entryram [8] = 0x4c ;
     mov    a, #04CH ; 76
     mov    [hl + 8], a
; line 16 : entryram [9] = 0x4c ;
     mov    [hl + 9] ,a
; line 17 : entryram [10] = 0x00 ;
     mov    a, #00H          ;0
     mov    [hl + 10] ,a
; line 18 : /* Set convergence time data */
; line 19 : entryram [11] = 0x01 ;
     inc    a
     mov    [hl + 11] ,a
; line 20 : entryram [12] = 0x3d ;
     mov    a, #03DH ; 61
     mov    [hl + 12] ,a
; line 21 : /* Call _FlashSetEnv subroutine */
; line 22 : ret = _ _FlashSetEnv (entryram) ;
     mov    c, #01H        ; 1
     call   !08100H
     mov    a, b
     mov    !_ret, a
            .
            .
            .
     pop    psw       ; Return to current register bank         ; Compiler automatically generates
ret                                                             ;  this line also
```

_ _ flashf Function _ _ flashf

RESTRICTIONS

- Functions other than **11.5 (42) On-chip firmware self-programming subroutine direct call function** cannot be called from the **_ _flashf** function.
- Only **char/unsigned char/int/unsigned int/short/unsigned short/pointer** type of 1 argument can be defined for a function argument.
- Only **char/unsigned char/int/unsigned int/short/unsigned short/pointer** type can be defined for automatic variables.
- Only a maximum of 6 bytes can be defined for argument and automatic variables combined.
- A **long** type operation cannot be performed.

COMPATIBILITY

<From another C compiler to this C compiler>

- No modifications are needed as long as the keyword **_ _ flashf** is not used.
- To change to the **_ _ flashf** function, modify according to the description method above.

<From this C compiler to another C compiler>

- Compatibility can be maintained with **#define** (Refer to **11.6 Modifications of C Source**).

(44) 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** and **memset** with direct inline expansion instead of function call.
- When there is no **#pragma** directive, the code that calls the standard library functions is generated.

EFFECT

- Compared with when a standard library function is called, the execution speed is improved.
- Object code is reduced if a constant is specified for the specified character number.

USAGE

- The function is described in the source in the same format as a function call.
- The following items can be described before **#pragma inline**.
  - Comments
  - Other **#pragma** directives
  - Preprocess directives that do not generate variable definitions/references or function definitions/references

EXAMPLE

(C source)

```
#pragma inline
char ary1[100], ary2[100];
void main()
{
     memset(ary1, 'A', 50);
     memcpy(ary1, ary2, 50);
}
```

Memory Manipulation Function #pragma inline

(Output object of compiler)

When **-SM** is not specified

```
_main:
        push     hl
;line    5:      memset(ary1, 'A', 50);
        movw     de,#_ary1
        mov      a, #041H  ; 65
        mov      c, #032H  ; 50
        mov      [de], a
        incw     de
        dbnz     c, $$-2
;line    6:      memcpy(ary1, ary2, 50);
        movw     de, #_ary1
        movw     hl, #_ary2
        mov      c, #032H  ; 50
        mov      a, [hl]
        mov      [de], a
        incw     de
        incw     hl
        dbnz     c, $$-4
;line    7:      }
        pop      hl
        ret
```

Memory Manipulation Function #pragma inline

When **-SM** is specified

```
_main:
        push     de
;line    5:      memset(ary1, 'A', 50);
        movw     hl,#_ary1
        mov      a, #041H  ; 65
        mov      c, #032H  ; 50
        mov      [hl], a
        incw     hl
        dbnz     c, $$-2
;line    6:      memcpy(ary1, ary2, 50);
        movw     hl, #_ary1
        movw     de, #_ary2
        mov      c, #032H  ; 50
        mov      a, [de]
        mov      [hl], a
        incw     de
        incw     hl
        dbnz     c, $$-4
;line    7:      }
        pop      de
        ret
```

COMPATIBILITY

<From another C compiler to this C compiler>

- Modification is not needed if the memory manipulation function is not used.
- When changing the memory manipulation function, use the method above.

<From this C compiler to another C compiler>

- The **#pragma inline** directive should be deleted or delimited using **#ifdef**.

(45) 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.

Address Specification Range	Secured Area Range	Duplication Check Range
0x80 to 0xffff	0xfd00 to 0xfeff	0xf000 to 0xfeff

- If the address specification is outside the address specification range, an error is output.
- If the allocation address of a variable declared by **_ _directmap** is duplicated and is within the duplication check range, a **W0762** warning message is output and the name of the duplicated variable is displayed.
- If the address specification range is inside the saddr area, the **_ _sreg** declaration is made automatically and the **saddr** instruction is generated.
- If **char/unsigned char/short/unsigned short/int/unsigned int/long/unsigned long** type variables declared by **_ _directmap** are bit referenced, **sreg/_ _sreg** must be specified along with **_ _directmap**. If they are not, an error occurs.

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;

- 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;

- To generate the **saddr** instruction in a module in which a **_ _directmap** external variable allocated inside the **saddr** area is referenced, **_ _sreg** must be used together to make extern**_ _sreg** Type name Variable name;.

EXAMPLE

(C source)

```
_ _directmap char c = 0xfe00;
_ _directmap _ _sreg char d = 0xfe20;
_ _directmap _ _sreg char e = 0xfe21;
_ _directmap struct x {
        char a;
        char b;
} xx = {0xfe30};
void main()
{
        c = 1;
        d = 0x12;
        e.5 = 1;
        xx.a = 5;
        xx.b = 10;
}
```

Absolute Address Allocation Specification __directmap

(Output object)

```
        PUBLIC  _c
        PUBLIC  _d
        PUBLIC  _e
        PUBLIC  _xx
        PUBLIC  _main
_c      EQU     0FE00H          ; Addresses for variables declared by __directmap
_d      EQU     0FE20H          ; are defined by EQU
_e      EQU     0FE21H          ;
_xx     EQU     0FE30H          ;
        EXTRN   __mmfe00        ; EXTRN output for linking secured area modules
        EXTRN   __mmfe20        ;
        EXTRN   __mmfe21        ;
        EXTRN   __mmfe30        ;
        EXTRN   __mmfe31        ;
@@CODE CSEG
_main:
;line    10:    c = 1;
        mov     a,#01H ;1
        mov     !_c,a
;line    11:    d = 0x12;
        mov     _d,#012H        ; saddr instruction output because address specified in saddr area
;line    12:    e.5 = 1;
        set1    _e.5            ; Bit manipulation possible because __sreg also used
;line    13:    xx.a = 5;
        mov     _xx,#05H        ; saddr instruction output because address specified in saddr area
;line    14:    xx.b = 10;
        mov     _xx+1,#0AH      ; saddr instruction output because address specified in saddr area
;line    15:    }
        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 **short/unsigned short/int/unsigned int/long/unsigned long** type variables are allocated to odd addresses, the correct code will be generated in the file declared by **_ _directmap**, but illegal code if these variables are referenced by an **extern** declaration from an external file.
- 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>

- No modification is necessary if the keyword **_ _directmap** is not used.
- To change to the **_ _directmap** variable, modify according to the description method above.

<From this C compiler to another C compiler>

- Compatibility can be attained using **#define** (refer to **11.6 Modifications of C Source** for details).
- When the **_ _directmap** is being used as the absolute address allocation specification, modify according to the specifications of each compiler.

(46) Static model expansion specification

Static Model Expansion Specification -ZM

FUNCTION

- The 8-byte **saddr** area of **_@NRAT00** to **_@NRAT07** is secured as area reserved by the compiler for arguments and work.
- Temporary variables can be used by declaring **_ _temp** for arguments and automatic variables (refer to **11.5 (47) Temporary variables** for details).
- The number of argument declarations that can be described ranges from 3 to 6 for int-sized variables and 3 to 9 for char-sized variables. The 4th and subsequent arguments are set by the calling side to the area of **_@NRAT00** to **_@NRAT05** and copied by the called side to a separate area. However, if **_ _temp** has been declared for a leaf function or an argument, the called side will not copy the argument, and the **_@NRATxx** area where the argument was set will be used as is.
- Structures and unions that are 2 bytes or smaller can be described for arguments.
- Structures and unions can be described for function return values. If the structures and unions are 2 bytes or smaller, the value will be returned. If 3 bytes or larger the return value will be stored in a static area secured for storing return values and returned to the top address of that area.
- The 8-byte area of **_@NRAT00** to **_@NRAT07** is also used as the leaf function shared area. In shared-area allocation, the 8-byte area of **_@NRAT00** to **_@NRAT07** is allocated to first, and then the **_@KREGxx** area secured by specifying the **-SM** option.
- Arrays, unions, and structures can also be allocated to **_@NRATxx** and **_@KREGxx**, provided their size fits into the **_@KREGxx** area secured by specifying **_@NRATxx** and **-SM**.
- Interrupt functions that are targeted for saving are shown in **Table 11-17** below.

Table 11-17 Interrupt Functions Targeted for Saving

Restore/Save Area	NO BANK	With Function Call				Without Function Call			
		-ZM1		-ZM2		-ZM1		-ZM2	
		Stack	RBn	Stack	RBn	Stack	RBn	Stack	RBn
Registers used	×	×	×	×	×	√	×	√	×
All registers	×	√	×	√	×	×	×	×	×
Entire _@NRATxx area	×	√	√	√	√	×	×	×	×
Entire _@KREGxx area	×	√	√	×	×	×	×	×	×
_@KREGxx area used	×	×	×	√	√	×	×	√	√

Stack: Stack use specification
RBn: Register bank specification
√: Saved
×: Not saved

Static Model Expansion Specification -ZM

Note, however, that when **#pragma interrupt** is specified, the interrupt functions that are targeted for saving can be limited by specifying as follows.

SAVE_R (save/restore targets limited to registers)

SAVE_RN (save/restore targets limited to registers and **_@NRATxx**).

- The only difference between the **-ZM1** and **-ZM2** options is in the treatment of the **_@KREGxx** area secured by specifying **-SM**.
  When the **-ZM1** option is specified, the **_@KREGxx** area is only used for leaf function shared area.
  When the **-ZM2** option is specified, the **_@KREGxx** area is saved/restored and arguments and automatic variables are allocated there (compatibility with the **-QR** option in the normal model).
- If the **-ZM** option is specified when the **-SM** option has not been specified, a **W0055** warning message is output and the **-ZM** option specification is disregarded.

EFFECT

Restrictions on existing static models can be relaxed, improving descriptiveness.

USAGE

Specify the -ZM option along with the -SM option when compiling.

EXAMPLE 1

(C source)

```
char func1(char a, char b, char c, char d, char e);
char func2(char a, char b, char c, char d);
void main()
{
        char a = 1, b = 2, c = 3, d = 4, e = 5, r;
        r = func1(a, b, c, d, e);
}
char func1(char a, char b, char c, char d, char e)
{
        char r;

        r = func2(a, b, c, d);
        return e + r;
}
char func2(char a, char b, char c, char d)
{
        return a + b + c + d;
}
```

Static Model Expansion Specification -ZM

(Output object)

When **-SM8**, **-ZM1**, and **-QC** are specified

```
_main:
;line    5:   char a = 1, b = 2, c = 3, d = 4, e = 5, r;
       mov     a,#01H ;1
       mov     !L0003,a       ; a
       inc     a
       mov     !L0004,a       ; b
       inc     a
       mov     !L0005,a       ; c
       inc     a
       mov     !L0006,a       ; d
       inc     a
       mov     !L0007,a       ; e
;line    6:
;line    7:   r = func1(a, b, c, d, e);
       mov     _@NRAT01,a     ; 5th argument set in saddr area for passing arguments
       mov     a,!L0006       ; d
       mov     _@NRAT00,a     ; 4th argument set in saddr area for passing arguments
       mov     a,!L0005       ; c
       mov     h,a
       mov     a,!L0004       ; b
       mov     b,a
       mov     a,!L0003       ; a
       call    !_func1
       mov     !L0008,a       ; r
;line    8:}
       ret
;line    9:   char func1(char a, char b, char c, char d, char e)
;line    10:{
_func1:
       mov     !L0011,a
       mov     a, b
       mov     !L0012,a
       mov     a,h
       mov     !L0013,a
       mov     a,_@NRAT00     ; Copied to static area
       mov     !L0014,a       ;
       mov     a,_@NRAT01     ; Copied to static area
       mov     !L0015,a
```

Static Model Expansion Specification -ZM

(Output object …continued)

```
;line    11:  char r;
;line    12:
;line    13:  r = func2(a, b, c, d)
       mov     a,!L0014       ; d
       mov     _@NRAT00,a     ; 4th argument set in saddr area for passing arguments
       mov     a,!L0013       ; c
       mov     h,a
       mov     a,!L0012       ; b
       mov     b,a
       mov     a,!L0011       ; a
       call    !_func2
       mov     !L0016,a       ; r
;line    14:  return e + r;
       add     a,!L0015       ; e
L0010:
;line    15:}
       ret
;line    16:  char func2(char a, char b, char c, char d)
;line    17:{
_func2:
       mov     _@NRAT01,a
       mov     a,b
       mov     _@NRAT02,a
       mov     a,h
       mov     _@NRAT03,a
;line    18:  return a + b + c + d;
       mov     a,_@NRAT01     ; a
       add     a,_@NRAT02     ; b
       add     a,_@NRAT03     ; c
       add     a,_@NRAT00     ; d  _@NRAT00 used as is for leaf function
L0018:
;line    19:}
       ret
```

Static Model Expansion Specification -ZM

When **-SM8**, **-ZM2**, and **-QC** are specified

```
    @@CODE CSEG
    _main:
           movw    ax,_@KREG10    ;
           push    ax             ; Area of _@KREG10 to _@KREG15 saved
           movw    ax,_@KREG12    ;
           push    ax             ;
           movw    ax,_@KREG14    ;
           push    ax             ;
    ;line     5:   char a = 1, b = 2, c = 3, d = 4, e = 5, r;
           mov     _@KREG15,#01H  ; a, 1  Variables allocated to _@KREG11 to _@KREG15
           mov     _@KREG14,#02H  ; b, 2
           mov     _@KREG13,#03H  ; c, 3
           mov     _@KREG12,#04H  ; d, 4
           mov     _@KREG11,#05H  ; e, 5
    ;line     6:
    ;line     7:   r = func1(a, b, c, d, e);
           mov     a,_@KREG11     ; e
           mov     _@NRAT01,a     ; 5th argument set in saddr area for passing arguments
           mov     a,_@KREG12     ; d
           mov     _@NRAT00,a     ; 4th argument set in saddr area for passing arguments
           mov     a,_@KREG13     ; c
           mov     h,a
           mov     a,_@KREG14     ; b
           mov     b,a
           mov     a,_@KREG15     ; a
           call    !_func1
           mov     _@KREG10,a     ; r
    ;line     8:}
           pop     ax             ;
           movw    _@KREG14,ax    ; Area of _@KREG10 to _@KREG15 restored
           pop     ax             ;
           movw    _@KREG12,ax    ;
           pop     ax             ;
           movw    _@KREG10,ax    ;
           ret
    ;line     9:   char func1(char a, char b, char c, char d, char e)
```

Static Model Expansion Specification -ZM

(Output object …continued)

```
;line    10:{
_func1:
      mov     _@NRAT06,a      ; Register a saved
      movw    ax,_@KREG10     ;
      push    ax              ; Area of _@KREG10 to _@KREG15 saved
      movw    ax,_@KREG12     ;
      push    ax              ;
      movw    ax,_@KREG14     ;
      push    ax              ;
      mov     a,_@NART06      ; Register a restored
      mov     _@KREG15,a
      movw    ax,bc
      mov     _@KREG14,a
      movw    ax,hl
      mov     _@KREG13,a
      mov     a,_@NART00      ; Copied to _@KREG12
      mov     _@KREG12,a
      mov     a,_@NART01      ; Copied to _@KREG11
      mov     _@KREG11,a
;line    11:  char r;
;line    12:
;line    13:  r = func2(a, b, c, d)
      mov     a,_@KREG12      ; d
      mov     _@NRAT00,a      ; 4th argument set in saddr area for passing arguments
      mov     a,_@KREG13      ; c
      mov     h,a
      mov     a,_@KREG14      ; b
      mov     b,a
      mov     a,_@KREG15      ; a
      call    !_func2
      mov     _@KREG10,a      ; r
;line    14:  return e + r;
      add     a,_@KREG11      ; e
L0004:
```

Static Model Expansion Specification -ZM

(Output object …continued)

```
;line    15:}
        movw    hl,ax           ; Register a saved
        pop     ax              ;
        movw    _@KREG14,ax     ; Area of _@KREG10 to _@KREG15 restored
        pop     ax              ;
        movw    _@KREG12,ax     ;
        pop     ax              ;
        movw    _@KREG10,ax     ;
        movw    ax,hl           ;  Register a restored
        ret
;line    16:  char func2(char a, char b, char c, char d)
;line    17:{
_func2:
        mov     _@NRAT01,a
        mov     a,b
        mov     _@NRAT02,a
        mov     a,h
        mov     _@NRAT03,a
;line    18:  return a + b + c + d;
        mov     a,_@NRAT01      ; a
        add     a,_@NRAT02      ; b
        add     a,_@NRAT03      ; c
        add     a,_@NRAT00      ; d  _@NRAT00 used as is for leaf function
L0006:
;line    19:}
        ret
```

Static Model Expansion Specification -ZM

EXAMPLE 2

(C source)

```
_ _sreg struct x {
        unsigned char a;
        unsigned char b:1;
        unsigned char c:1;
} xx,yy;
_ _sreg struct y {
        int a;
        int b;
} ss, tt;
struct x func1(struct x);
struct y func2();
void main()
{
        yy = func1(xx);
        tt = func2();
}
struct x func1(struct x aa)
{
        aa.a = 0x12;
        aa.b = 0;
        aa.c = 1;
        return aa;
}
struct y func2()
{
        return tt;
}
```

Static Model Expansion Specification -ZM

(Output object)

When **-SM** and **-ZM** are specified

```
@@CODE CSEG
_main:
;line     14: yy = func1(xx);
        movw   ax,_xx
        call   !_func1
        movw   _yy,ax
;line     15: tt = func2();
        call   !_func2
        movw   hl,ax
        push   de
        movw   de,#_tt
        mov    c,#04H ;4
        mov    a,[hl]
        mov    [de],a
        incw   hl
        incw   de
        dbnz   c,$$-4
        pop    de
;line     16: }
        ret
;line     17: struct x func1(struct x aa)
;line     18: {
_func1:
        movw   _@NRAT00,ax
;line     19: aa.a = 0x12;
        mov    _@NRAT00,#012H  ; aa,18
;line     20: aa.b = 0;
        clr1   _@NRAT01.0
;line     21: aa.c = 1;
        set1   _@NRAT01.1
;line     22: return aa;
        movw   ax,_@NRAT00      ; aa  Value returned because 2 bytes or smaller
;line     23:}
        ret
;line     24: struct y func2()
;line     25: {
```

Static Model Expansion Specification -ZM

(Output object …continued)

```
    ;line      26: return tt;
            movw   hl,#_tt          ; Return value copied to secured static area because
            push   de               ; 3 bytes or larger
            movw   de,#L0007
            mov    c,#04H ;4
            mov    a,[hl]
            mov    [de],a
            incw   hl
            incw   de
            dbnz   c,$$-4
            pop    de
            movw   ax,#L0007        ; Returned top address of static area
    ;line      27: }
            ret
```

COMPATIBILITY

<From another C compiler to this C compiler>

- The source program need not be modified.

<From this C compiler to another C compiler>

- The source program need not be modified.

(47) Temporary variables

Temporary Variables _ _temp

FUNCTION

- Arguments and automatic variables are allocated to the area of **_@NRAT00** to **_@NRAT07**, regardless of whether they correspond to a leaf function. If arguments and automatic variables are not allocated to the area of **_@NRAT00** to **_@NRAT07** they will be treated in the same way as when **_ _temp** is not declared.
- The values of arguments and automatic variables declared by **_ _temp** are discarded upon a function call.
- **_ _temp** cannot be declared for external and static variables.
- If **_ _sreg** is declared as well, char/unsigned char/short/unsigned short/int/unsigned int variables can be bit manipulated.
- If **_ _temp** is declared when the **-SM** and **-ZM** options have not been specified, a **W0339** warning message is output and the **_ _temp** declaration in the file is disregarded.

EFFECT

- Because arguments and automatic variables declared by **_ _temp** share the area of **_@NRAT00** to **_@NRAT07**, an argument and automatic variable area can be reserved.
- If the sections containing arguments and those containing automatic variables are clearly identified and the **_ _temp** declaration is applied to variables that do not require a guaranteed value match before and after a function call, memory can be reserved.

USAGE

Specify the **-SM** and **-ZM** options during compilation and declare **_ _temp** for arguments and automatic variables.

EXAMPLE

(C source)

```
void func1(_ _temp char a, char b, char c, _ _sreg _ _temp char d);
void func2(char a);
void main()
{
        func1(1, 2, 3, 4);
}
void func1(_ _temp char a, char b, char c, _ _sreg _ _temp char d)
{
        _ _temp char r;

        d.1 = 0;
        r = a + b + c + d;
        func2(r);
}
void func2(char r)
{
        int a = 1, b = 2;
        r++;
}
```

Temporary Variables _ _temp

(Output object)

When **-SM**, **-ZM**, and **-QC** are specified

```
@@CODE CSEG
_main:
;line     5:  func1(1, 2, 3, 4);
       mov    a,#04H    ; 4
       mov    _@NRAT00,a
       mov    h,#03H    ; 3
       mov    b,#02H    ; 2
       mov    a,#01H    ; 1
       call   !_func1
;line     6:  }
       ret
;line     7:  void func1(_ _temp char a, char b, char c, _ _sreg _ _temp char d)
;line     8:  {
_func1:
       mov    _@NRAT01,a        ; Allocated to _@NRAT01
       mov    a,b
       mov    !L0005,a
       mov    a,h
       mov    !L0006,a
                                ; Argument allocated to _@NRAT00 is unchanged
;line     9:  _ _temp char r;
;line     10:
;line     11: d.1 = 0;
       clr1   _@NRAT00.1        ; Bit manipulation possible
;line     12: r = a + b + c + d;
       mov    a,_@NRAT01        ; a
       add    a,!L0005          ; b
       add    a,!L0006          ; c
       add    a,_@NRAT00        ; d
       mov    _@NRAT02,a        ; r  _@NRAT02 used
;line     13: func2(r);
       call   !_func2
;line     14: {
                                ; Values of _@NRAT00 to _@NRAT02 changed after return
       ret
;line     15: void func2(char r)
```

Temporary Variables __temp

(Output object …continued)

```
;line      16: {
_func2:
        mov    _@NRAT00,a
;line      17: int a = 1, b = 2;
        movw   _@NRAT02,#01H   ; a,1
        movw   _@NRAT04,#02H   ; b,2
;line      18: r++;
        inc    _@NRAT00
;line      19: }
        ret
```

RESTRICTIONS

If there are 3 arguments or fewer when a function is called, arguments and automatic variables declared by **__temp** can be described for the arguments at function call. If there are 4 or more arguments, because the values of the arguments could be discarded during argument evaluation, values described cannot be guaranteed.

COMPATIBILITY

<From another C compiler to this C compiler>

- Modification is not necessary if the reserved word **__temp** is not used.
- To change to a temporary variable, modify according to the description 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). This modification means that the **__temp** variable is treated as an ordinary variable.

(48) Library supporting prologue/epilogue

Library Supporting Prologue/Epilogue -ZD

FUNCTION

- A specified pattern of the prologue/epilogue code can be replaced with a library call.
- Number of callt entries that users can use is reduced two in the case of a normal model and up to ten in the case of a static model.
- The library replacement patterns in the case of a normal model are as follows.
  HL,**_@KREGxx** save/copy, stack frame secure → **callt [@@cprep2]**
  HL,**_@KREGxx** restore, stack frame release → **callt [@@cdisp2]**

- In the case of a static model, arguments are allocated to **_@NRATxx** and **_@KREGxx** so that the first 3 arguments accord with the patterns described below. When **char** and **int** are mixed, the allocation interval is adjusted so that it accords with the patterns of multiple **int** type arguments.
- The library replacement pattern in the case of a static model is as follows.

(For char 2 arguments)

```
mov     _@NRAT00,a      →       callt [@@nrcp2]
mov     a,b
mov     _@NRAT01,a

mov     _@KREG15,a      →       callt [@@krcp2]
mov     a,b
mov     _@KREG14,a
```

(For char 3 arguments)

```
mov     _@NRAT05,a      →       callt [@@nrcp3]
mov     a,b
mov     _@NRAT06,a
mov     a,h
mov     _@NRAT07,a

mov     _@KREG15,a      →       callt [@@krcp3]
mov     a,b
mov     _@KREG14,a
mov     a,h
mov     _@KREG13,a

mov     _@NRAT06,a      →       call !@@nkrc3
mov     a,b
mov     _@NRAT07,a
mov     a,h
mov     _@KREG15,a
```

Library Supporting Prologue/Epilogue -ZD

(For int 2 arguments)

```
movw    _@NRAT00,ax     →       callt [@@nrip2]
movw    ax,bc
movw    _@NRAT02,ax

movw    _@KREG14,ax     →       callt [@@krip2]
movw    ax,bc
movw    _@KREG12,ax
```

(For int 3 arguments)

```
movw    _@NRAT02,ax     →       callt [@@nrip3]
movw    ax,bc
movw    _@NRAT04,ax
movw    ax,hl
movw    _@NRAT06,ax

movw    _@KREG14,ax     →       callt [@@krip3]
movw    ax,bc
movw    _@KREG12,ax
movw    ax,hl
movw    _@KREG10,ax

movw    _@NRAT04,ax     →       call !@@nkri31
movw    ax,bc
movw    _@NRAT06,ax
movw    ax,hl
movw    _@KREG14,ax

movw    _@NRAT06,ax     →       call !@@nkri32
movw    ax,bc
movw    _@KREG14,ax
movw    ax,hl
movw    _@KREG12,ax
```

Library Supporting Prologue/Epilogue -ZD

(For save/restore)

```
_@NRAT00 to _@NRAT07 save    →    callt [@@nrsave]

_@NRAT00 to _@NRAT07 restore →    callt [@@nrload]

_@KREG14 to 15 save          →    call !@@krs02

_@KREG12 to 15 save          →    call !@@krs04
                             →    call !@@krs04i

_@KREG10 to 15 save          →    call !@@krs06
                             →    call !@@krs06i

_@KREG08 to 15 save          →    call !@@krs08
                             →    call !@@krs08i

_@KREG06 to 15 save          →    call !@@krs10
                             →    call !@@krs10i

_@KREG04 to 15 save          →    call !@@krs12
                             →    call !@@krs12i

_@KREG02 to 15 save          →    call !@@krs14
                             →    call !@@krs14i

_@KREG00 to 15 save          →    call !@@krs16
                             →    call !@@krs16i

_@KREG14 to 15 restore       →    call !@@krl02

_@KREG12 to 15 restore       →    call !@@krl04
                             →    call !@@krl04i

_@KREG10 to 15 restore       →    call !@@krl06
                             →    call !@@krl06i

_@KREG08 to 15 restore       →    call !@@krl08
                             →    call !@@krl08i

_@KREG06 to 15 restore       →    call !@@krl10
                             →    call !@@krl10i

_@KREG04 to 15 restore       →    call !@@krl12
                             →    call !@@krl12i

_@KREG02 to 15 restore       →    call !@@krl14
                             →    call !@@krl14i

_@KREG00 to 15 restore       →    call !@@krl16
                             →    call !@@krl16i
```

Library Supporting Prologue/Epilogue -ZD

EFFECT

By replacing prolog and epilog code with a library, object code can be shortened.

USAGE

Specify the **-ZD** option during compilation.

EXAMPLE 1

(C source)

```
int func1(int a, int b, int c);
int func2(int a, int b, int c);
void main()
{
        int r;

        r = func1(1, 2, 3);
}
int func1(int a, int b, int c)
{
        return func2(a+1, b+1, c+1);
}
int func2(int a, int b, int c)
{
        return a+b+c;
}
```

Library Supporting Prologue/Epilogue -ZD

(When **-SM8**, **-ZM2D**, and **-QC** are specified)

```
    @@CODE CSEG
    _main:
            movw     ax,_@KREG14
            push     ax
    ;line       5:   int r;
    ;line       6:
    ;line       7:   r = func1(1, 2, 3);
            movw     hl,#03H ; 3
            movw     bc,#02H ; 2
            movw     ax,#01H ; 1
            call     !_func1
            movw     _@KREG14,ax      ; r
    ;line       8:   }
            pop      ax
            movw     _@KREG14,ax
            ret
    ;line       9:   int func1(int a, int b, int c)
    ;line       10:  {
    _func1:
            call     !@@krs06
            callt    [@@krip3]
    ;line       11:  return func2(a+1, b+1, c+1);
            movw     ax,_@KREG10      ; c
            incw     ax
            movw     hl,ax
            movw     ax,_@KREG12      ; b
            incw     ax
            movw     bc,ax
            movw     ax,_@KREG14      ; a
            incw     ax
            call     !_func2
    L0004:
    ;line       12:  }
            call     !@@krl06
            ret
    ;line       13:  int func2(int a, int b, int c)
    ;line       14:  {
    _func2:
            callt    [@@nrip3]
```

Library Supporting Prologue/Epilogue -ZD

(When **-SM8**, **-ZM2D**, and **-QC** are specified) (continued)

```
;line       15:  return a+b+c;
        movw     ax, _@NRAT02    ; a
        xch      a,x
        add      a, _@NRAT04     ; b
        xch      a,x
        addc     a, _@NRAT05     ; b
        xch      a,x
        add      a, _@NRAT06     ; c
        xch      a,x
        addc     a, _@NRAT07     ; c
L0006:
;line       16:  }
        ret
```

EXAMPLE 2

(C source)

```
int func(register int a, register int b);
void main()
{
        register int a = 1, b = 2, c = 3,r;

        r = func(a, b);
}
int func(register int a, register int b)
{
        register int r;

        r = a + b;
        return r;
}
```

Library Supporting Prologue/Epilogue -ZD

(Output object)

When **-QR** and **-ZD** are specified

```
    @@CODE CSEG
    _main:
            movw     de,#0300H
            callt    [@@cprep2]
    ;line        4:  register int a = 1, b = 2, c = 3,r;
            movw     hl,#01H ;1
            movw     _@KREG00,#02H    ; b,2
            movw     _@KREG02,#03H    ; c,3
    ;line        5:
    ;line        6:  r = func(a, b);
            movw     ax,_@KREG00      ; b
            push     ax
            movw     ax,hl
            call     !_func
            pop      ax
            movw     ax,bc
            movw     _@KREG04,ax      ; r
    ;line        7:  }
            movw     ax,#0300H
            callt    [@@cdisp2]
            ret
    ;line        8:  int func(register int a, register int b)
    ;line        9:  {
    _func:
            movw     de,#0C940H
            callt    [@@cprep2]
    ;line       10:  register int r;
    ;line       11:
    ;line       12:  r = a + b;
            movw     ax,hl
            xch      a,x
            add      a,_@KREG12       ; a
            xch      a,x
            addc     a,_@KREG13       ; a
            movw     _@KREG00,ax      ; r
    ;line       13:  return r;
            movw     bc,ax
    L0004:
    ;line       14:  }
            movw     ax,#0C940H
            callt    [@@cdisp2]
            ret
```

Library Supporting Prologue/Epilogue -ZD

RESTRICTIONS

- The optimization specification option **-QL4** cannot be specified at the same time as the **-ZD** option. If it is specified, a **W0052** warning message is output and the **-QL4** option is replaced with the **-QL3** option and processed.
- The old function interface compatibility option **-ZO** cannot be specified at the same time as the **-ZD** option. If it is specified, a **W0053** warning message is output and the **-ZD** option is disregarded.
- The flash area allocation specification option **-ZF** cannot be specified at the same time as the **-ZD** option. If it is specified, a **W0054** warning message is output and the **-ZD** option is disregarded.

CAUTION

The argument copy pattern in the case of a static model will be pattern-matched only when **register** has not been specified for any of the first 3 arguments or **_ _temp** has been specified for all of the first 3 arguments. Therefore, because pattern matching will not be performed if the **-QV** option is specified or if **register/_ _temp** are partially specified for the first 3 arguments, it will no longer be possible to replace the **-ZD** option specification.

COMPATIBILITY

<From another C compiler to this C compiler>

- The source program need not be modified.
- To replace the prologue/epilogue code with a library, modify the source program according to the description method above.

<From this C compiler to another C compiler>

- The source program need not be modified.

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 78K0 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** preprocessor directive, the C source must be modified. The method and extent of modifications to the C source depend on the specifications of the other C compiler.
- Extended specifications
  If the other C compiler has extended specifications such as addition of keywords, the C source must be modified. The method and extent of modifications to the C source depend on the specifications of the other C compiler.

Note **#pragma** is one of the preprocessing directives supported by ANSI. The character string following the **#pragma** is identified as a directive to the compiler. If the compiler does not support this directive, the **#pragma** directive is ignored and the compile will be continued until it properly ends.

<From 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 invalidating them with **#ifdef.**

EXAMPLE

<1> To invalidate a keyword (Same applies to **callf**, **sreg**, **noauto**, and **norec**, etc.)

```
#ifndef _ _K0_ _
        #define callt        /* makes callt as ordinary function */
#endif
```

<2> To change from one type to another

```
#ifndef _ _K0_ _
        #define bit char     /* changes bit type to char type variable */
#endif
```

11.7 Function Call Interface

The following will be explained about the interface between functions at function call.

1. Return value (common in all the functions)

2. Ordinary function call interface
   (1) Passing arguments
   (2) Location and order of storing arguments
   (3) Location and order of storing automatic variables

3. **noauto** function call interface
   (1) Passing arguments
   (2) Location and order of storing arguments
   (3) Location and order of storing automatic variables

4. **norec** function call interface
   (1) Passing arguments
   (2) Location and order of storing arguments
   (3) Location and order of storing automatic variables

5. Static model function call interface
   (1) Passing arguments
   (2) Location and order of storing arguments
   (3) Location and order of storing automatic variables

6. 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 11-18**.

Table 11-18 Location of Storing Return Value

Type \ Model	Normal Model	Static Model
1-byte integer	BC	A
2-byte integer		AX
4-byte integer	BC (Lower) DE (Upper)	Not supported
Pointer	BC	AX
Structure, union	BC (if copied to the area specific to the function, the start address of the structure or union)	Not supported
1 bit	CY (carry flag)	CY (carry flag)
Floating-point number (**float** type)	BC (Lower) DE (Upper)	Not supported
Floating-point number (**double** type)	BC (Lower) DE (Upper)	Not supported

11.7.2 Ordinary function call interface

When all the arguments are allocated to registers and there is not an automatic variable, the ordinary function call interface is the same as **noauto** function call interface.

(1) Passing arguments

- There are two types of arguments; arguments that are allocated to a register and normal arguments.
- An argument that is allocated to a register is an argument that has undergone register declaration and is allocated to a register or **_@KREGxx** as long as an allocatable register and **_@KREGxx** exist. However, arguments are allocated to **_@KREGxx** only when **-QR** is specified. Arguments that are allocated to a register or **_@KREGxx** are referred to as register arguments hereafter.
- Refer to **APPENDIX A LIST OF LABELS FOR saddr AREA** for **_@KREGxx**.
- The remaining arguments are allocated to a stack.

(a) When -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 argument and later are passed via a stack, and the first argument is passed via a register or stack.
- On the function definition side, arguments passed via register or stack are saved in the place where arguments are allocated.
- Register arguments are copied in a register or **_@KREGxx**. Even when the arguments are passed via registers, register copying is necessary since the registers on the function caller (passing side) are different to those on the function definition side (receiving side).
- Normal arguments are loaded on a stack. When an argument is passed via a stack, the area where the arguments are passed to becomes the area to which they are allocated.
- Saving and restoring registers to which arguments are allocated is performed on the function definition side.
- The location where the first argument is passed is shown in **Table 11-19**.

Table 11-19 Location Where the First Argument Is Passed (on the Function Call Side)

Type \ Option	Normal Model without **-ZO** Specification	Normal Model with **-ZO** Specification
1-byte data[Note] 2-byte data[Note]	**AX**	Passed via stack
3-byte data[Note]	**AX**, **BC**	Passed via stack
4-byte data[Note]	**AX**, **BC**	Passed via stack
Floating-point number (**float** type)	**AX**, **BC**	Passed via stack
Floating-point number (**double** type)	**AX**, **BC**	Passed via stack
Others	Passed via stack	Passed via stack

Note 1- to 4-byte data include structure, union, and pointer.

(b) When -ZO option is specified

- Not allocated to a register.
- **_@KREG12** to **15** are used for the **saddr** area.
- Register arguments are passed via **_@KREGxx** and normal arguments are passed via a stack.
- The area where the arguments are passed (stack) is used for the area to which arguments are allocated.
- **_@KREGxx,** to which arguments are allocated, is saved before the function call and is restored after the function call.

(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 arguments declared with registers and 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 -ZO option is not specified

- Saving and restoring registers to which arguments are allocated is performed on the function definition side.
- 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.
- The register arguments are copied to a register or **_@KREGxx**. Copying to **_@KREGxx** is performed only when **-QR** is specified. Even when the arguments are passed via registers, register copying is necessary since the registers on the function caller (passing side) are different to those on the function definition side (receiving side).
- On the function caller, both register arguments and normal arguments are passed using the same method.
  The second or later arguments are passed via a stack. The first argument is passed via a register and stack.
  Refer to **Table 11-19** for the place where the first argument is passed.

(Registers to be used)

HL

Arguments are not allocated to **HL** when there is a stack frame.

(saddr area to be used)

_@KREG12 to 15

(Allocation sequence)

- Registers

char type:	The sequence is **L-H**.
int, **short**, and **enum** type:	**HL**

- **saddr** area

char type:	The sequence is **_@KREG12**, **_@KREG13**, **_@KREG14**, and **_@KREG15**.
int, **short**, and **enum** type:	The sequence is **_@KREG12** to **13** and **_@KREG14** to **15**.
long, **float**, **double** type:	The sequence is **_@KREG12** to **13** (low-order)-**_@KREG14** to **15** (high-order).

(b) When -ZO option is specified

- Not allocated to a register
- **_@KREG12** to **15** are used for the saddr area (only when **-QR** is specified).
- Register arguments are passed via **_@KREGxx** and normal arguments are passed via a stack.
- The area where the arguments are passed (stack) on the function caller or function definition side is used as the area to which argument are allocated.
- **_@KREGxx**, to which arguments are allocated, is saved or restored before and after the function call.

(Allocation sequence)

- Arguments are not allocated to a register. The arguments are allocated to **_@KREG12** to **15** based on the declaration sequence.

(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 ones which are declared with registers and automatic variables with **-QV** is specified. They are allocated to register **_@KREGxx** as long as there are allocable registers and **_@KREGxx**. However, 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 register when there are excess registers after the allocation of register arguments.
- The automatic variables not allocated to a register are allocated to a stack.
- The save and restoration to registers and **_@KREGxx** to allocate automatic variables are performed on the function definition side.

(a) Automatic variable allocation sequence

The allocation sequence of automatic variables to **_@KREGxx** is as follows.

(Registers to be used)

HL

Arguments are not allocated to **HL** when there is a stack frame.

(saddr area to be used)

_@KREG00 to **11**

(Allocation sequence)

- Registers

char type:	The sequence is **L** and **H**.
int, **short**, and **enum** type:	**HL**

- **saddr** area

char type:	The sequence is **_@KREG00**, **_@KREG01** ..., and **_@KREG11**.
int, **short**, and **enum** type:	The sequence is **_@KREG00** to **01**, **_@KREG02** to **03** ... and **_@KREG10** to **11**.
long, **float**, **double** type:	The sequence is **_@KREG00** to **03**, **_@KREG04** to **07**, and **_@KREG08** to **11**.

- The automatic variables that are allocated to a stack are loaded on the stack in the sequence of declaration.

[Example]

(C source 1)

```
void func0 (register int, int) ;
void main ()
{
        func (0x1234, 0x5678) ;
}
void func (register int p1, int p2)
{
        register int r ;
        int a ;
        r = p2 ;
        a = p1 ;
}
```

(Output code)

When **-ZO** is specified

```
_main :
; line 4 :   func0 (0x1234, 0x5678) ;
     movw    ax, #05678H      ; 22136
     push    ax                          ; Argument passed via a stack
     movw    _@NRAT00, #01234H; 4660     ;
     movw    ax, _@KREG12                ;
     push    ax                          ; Saves the saddr area for the register argument
     movw    ax, _@NRAT00                ;
     movw    _@KREG12, ax                ; Allocate the register argument to _@KREG12
     call    !_func0                     ; Function call
     pop     ax                          ;
     movw    _@KREG12, ax                ; Restores the saddr area for the register argument
     pop     ax                          ; Argument passed via a stack
     ret
; line 6 :   void func0 (register int p1, int p2)
; line 7 :   {
_func0 :
     push    hl
     movw    ax, _@KREG00
     push    ax                          ; Saves the saddr area for the register variable
     push    ax                          ; Reserves the area for automatic variable a
```

(Output code ...continued)

```
        movw    ax, sp
        movw    hl, ax
    ; line  8 :  register int r ;
    ; line  9 :  int a ;
    ; line  10 : r = p2 ;
        mov     a, [hl + 10]     ; p2        ; Argument p2 passed via a stack
        xch     a, x
        mov     a, [hl + 11]     ; p2
        movw    _@KREG00, ax     ; r         ; Assign to the register variable _@KREG00
    ; line  11 : a = p1 ;
        movw    ax, _@KREG12     ; p1        ; Pass the register argument _@KREG12
        mov     [hl + 1]  a      ; a
        xch     a, x
        mov     [hl], a          ; a         ; Assign to automatic variable a
        pop     ax                           ; Releases the area for automatic variable a
        pop     ax
        movw    _@KREG00, ax                 ; Restores the saddr area for register variables
        pop     hl                           ;
        ret
```

(Output code)

When **-ZO** is not specified

```
    _main :
    ; line   4 : func0 (0x1234, 0x5678) ;
        movw    ax, #05678H      ; 22136
        push    ax                           ; Argument passed via a stack
        movw    ax, #01234H      ; 4660      ; The first argument that is passed via a register
        call    !_func0                      ; Function call
        pop     ax                           ; Argument passed via a stack
    ; line   5 : }
        ret
    ; line   6 : void func0 (register int p1, int p2)
    ; line   7 :{
    _func0 :
        push    hl
        xch     a, x
        xch     a, _@KREG12
        xch     a, x
        xch     a, _@KREG13                  ; Allocate register argument p1 to _@KREG12
```

(Output code …continued)

```
        push    ax                              ; Saves the saddr area for register argument
        movw    ax, _@KREG00
        push    ax                              ; Saves the saddr area for a register variable
        push    ax                              ; Reserves area for the automatic variable a
        movw    ax, sp
        movw    hl, ax
; line 8 :    register int r ;
; line 9 :    int a ;
; line 10 :   r = p2 ;
        mov     a, [hl + 10]      ; p2          ; Argument p2 passed via a stack
        xch     a, x
        mov     a, [hl + 11]      ; p2
        movw    _@KREG00, ax      ; r           ; Assigns to register variable _@KREG00
; line 11 :   a = p1 ;
        movw    ax, _@KREG12      ; p1          ; Register argument _@KREG12
        mov     [hl + 1] , a      ; a
        xch     a, x
        mov     [hl] , a          ; a           ; Assigns to automatic variable a
; line 12 :   }
        pop     ax                              ; Releases area of the automatic variable a
        pop     ax
        movw    _@KREG00, ax                    ; Restores the saddr area for a register variables
        pop     ax
        move    _@KREG12, ax                    ; Restores the saddr area for a register argument
        pop     hl
        ret
```

(C source 2)

```
void func1 (int, register int) ;
void main ()
{
        func1 (0x1234, 0x5678) ;
}
void func1 (int p1, register int p2)
{
        register int r ;
        int a ;
        r = p2 ;
        a = p1 ;
}
```

(Output code)

When **-ZO** is specified

```
_main :
; line 4 :   func1 (0x1234, 0x5678) ;
     movw    _@NRAT00, #05678H ; 22136
     movw    ax, #01234H       ; 4660
     push    ax                           ; Argument passed via a stack
     movw    ax, _@KREG12
     push    ax                           ; Saves the saddr area for the register argument
     movw    ax, _@NRAT00
     movw    _@KREG12, ax                 ; Assigns the register argument to _@KREG12
     call    !_func1                      ; Function call
     pop     ax
     movw    _@KREG12, ax                 ; Restores the saddr area for register argument
     pop     ax                           ; Argument passed via a stack
; line 5 :   }
     ret
; line 6 :   void func1 (int p1, register int p2)
; line 7 :   {
_func1 :
     push    hl
     movw    ax, _@KREG00
     push    ax                           ; Saves the saddr area for register variables
     push    ax                           ; Reserves the area for the automatic variable a
     movw    ax, sp
     movw    hl, ax
; line 8 :   register int r ;
; line 9 :   int a ;
; line 10 :  r = p2 ;
     movw    ax, _@KREG12      ; p2
     movw    _@KREG00, ax      ; r        ; Register variable _@KREG00
; line 11 :  a = p1 ;
     mov     a, [hl + 10]      ; p1       ; Argument p1 (lower-order) passed via a stack
     mov     [hl] , a          ; a        ; Automatic variable a (low-order)
     xch     a, x
     mov     a, [hl + 11]      ; p1       ; Argument p1 (high-order) passed via a stack
     mov     [hl + 1] , a      ; a        ; Releases the area for automatic variable a
; line 12 :  }
     pop     ax                           ; Restores the saddr area for register variable
     pop     ax
     movw    _@KREG00, ax                 ; Restore the saddr area for register variable
     pop     hl
     ret
```

(Output code)

When **-ZO** is not specified

```
_main :
; line 4 :   func1 (0x1234, 0x5678) ;
    movw    ax, #05678H     ; 22136
    push    ax                          ; Argument passed via a stack
    movw    ax, #01234H     ; 4660      ; The first argument that is passed via a register
    call    !_func1                     ; Function call
    pop     ax                          ; Argument passed via a stack
; line 5 :   }
    ret
; line 6 :   void func1 (int p1, register int p2)
; line 7 :   {
_func1 :
    push    hl
    push    ax                          ; Loads the first argument p1 on the stack
    movw    ax, _@KREG00
    push    ax                          ; Saves the saddr area for register variables
    movw    ax, _@KREG12
    push    ax                          ; Saves the saddr area for register arguments
    push    ax                          ; Reserves area for the automatic variable a
    movw    ax, sp
    movw    hl, ax
    mov     a, [hl + 12]                ; Argument p2 passed via a stack and received via
                                        ; the saddr area
    xch     a, x
    mov     a, [hl + 13]
    movw    _@KREG12, ax                ;  Allocates the register argument to _@KREG12.
; line 8 :   register int r ;
; line 9 :   int a ;
; line 10 :  r = p2 ;
    movw    ax, _@KREG12    ; p2
    movw    _@KREG00, ax    ; r         ; Register variable _@KREG00
; line 11 :  a = p1 ;
    mov     a, [hl + 6]     ; p1        ; Argument p1 (low-order) passed via a stack
                                        ; and received by a register
    mov     [hl] , a        ; a         ; Automatic variable a (low-order)
    xch     a, x
    mov     a, [hl + 7]     ; p1        ; Argument p1 (high-order) passed via a stack
                                        ; and received by a register
    mov     [hl + 1] , a    ; a         ; Automatic variable a (high-order)
; line 12 :  }
    pop     ax                          ; Releases area for the automatic variable a
    pop     ax
    movw    _@KREG12, ax                ; Restores the saddr area for register arguments
    pop     ax
    movw    _@KREG00, ax                ; Restores the saddr area for register variables
    pop     ax
    pop     hl
    ret
```

11.7.3 noauto function call interface (normal model only)

(1) Passing arguments

(a) When -ZO option is not specified (default)

- On the function caller, arguments are passed in the same way as in an ordinary function. Refer to **11.7.2 Ordinary function call interface**.
- On the function definition side, arguments passed via a register or stack are copied to a register as well as **_@KREG12** to **15**. Copying to **_@KREG12** to **15** is performed only when **-QR** is specified. Even when the arguments are passed via registers, register copying is necessary since the registers on the function caller (passing side) are different to those on the function definition side (receiving side).
- Saving and restoring registers to which arguments are allocated is performed on the function definition side.

(b) When -ZO option is specified

- Arguments are transferred via **_@KREG12** to **15** (only when **-QR** is specified).
- The area where arguments are passed on the function caller and the function definition side becomes the area to which arguments are allocated.
- **_@KREG12** to **15** to which arguments are allocated, is saved or restored before and after the function call.

(2) Location and order of storing arguments

- On the function definition side, all arguments are allocated to registers and **_@KREG12** to **15**. However, arguments are allocated to **_@KREG12** to **15** only when **-QR** is specified.
- If there are arguments that are not allocated to registers or **_@KREG12** to **15** an error will result.

(a) When -ZO option is not specified (default)

- On the function caller, arguments are passed in the same way as in an ordinary function (Refer to **11.7.2 Ordinary function call interface**).
- On the function definition side, the arguments passed via a register or stack are copied to a register as well as **_@KREG12** to **15**. Even when the arguments are passed via registers, register copying is necessary since the registers on the function caller (passing side) are different to those on the function definition side (receiving side).
- Saving and restoring registers to which arguments are allocated is performed on the function definition side.

(Allocation sequence)

- The allocation sequence is the same as in a ordinary function (Refer to **11.7.2 Ordinary function call interface**).

(b) When -ZO option is specified

- Arguments are transferred by **_@KREG12** to **15**.
- The area where the arguments are passed (stack) is used for the area to which argument are allocated.
- **_@KREG12** to **15** to which arguments are allocated, is saved or stored before and after the function call.

(Allocation sequence)

- The allocation sequence is the same as in a ordinary function (Refer to **11.7.2 Ordinary function call interface**).

(3) Location and order of storing automatic variables

(a) When -ZO option is not specified (default)

Automatic variables are allocated to registers and **_@KREG12** to **15**. However, automatic variables are allocated to **_@KREG12** to **15** only when **-QR** is specified. For **_@KREG12** to **15**, 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 **_@KREG12** to **15**.

If an automatic variable cannot be allocated to registers and **_@KREG12** to **15**, an error occurs.

The save and restoration of the register and **_@KREG12** to **15** to which automatic variables are allocated are performed on the function definition side.

(Allocation sequence)

- The order of allocating automatic variables to registers is the same as the order of allocating arguments.
- The automatic variables allocated to **_@KREG12** to **15** are allocated in the order of declaration.

(b) When -ZO option is specified

- Allocation cannot be performed because the automatic variables cannot be described.

[Example]

(C source)

```
noauto void func2 (int, int) ;
void main ()
{
        func2 (0x1234, 0x5678) ;
}
noauto void func2 (int p1, int p2)
{
  .
  .
  .
}
```

(Output code)

When **-ZO** is specified

```
    _main :
    ; line   4 : func2 (0x1234, 0x5678) ;
         movw    ax, _@KREG12
         push    ax                              ; Saves the saddr area for argument
         movw    _@KREG12, #05678H; 22136        ; Allocate the argument to _@KREG12
         push    hl                              ; Saves a register for the argument
         movw    hl, #01234H      ; 4660         ; Allocate the argument to HL
         call    !_func2                         ; Function call
         pop     hl                              ; Restores the register for arguments
         pop     ax
         movw    _@KREG12, ax                    ; Restores the saddr area for arguments
    ; line   5 : }
         ret
    ; line   6 : noauto void func2 (int p1, int p2)
    ; line   7 : {
    _func2 :
      .
      .
      .
         ret
```

(Output code)

When **-ZO** is not specified

```
    _main :
    ; line   4 : func2 (0x1234, 0x5678) ;
         movw    ax, #05678H      ; 22136
         push    ax                              ; Argument passed via a stack
         movw    ax, #01234H      ; 4660         ; The first argument that is passed via a register
         call    !_func2                         ; Function call
         pop     ax                              ; Argument passed via a stack
    ; line   5 : }
         ret
    ; line   6 : noauto void func2 (int p1, int p2)
    ; line   7 : {
    _func2 :
         push    hl                              ; Saves a register for the argument
         xch     a, x
         xch     a, _@KREG12                     ; Allocate the argument p1 to _@KREG12 (lower)
         xch     a, x
         xch     a, _@KREG13                     ; Allocate the argument p1 to _@KREG13 (higher)
```

(Output code …continued)

```
        push    ax                      ; Saves the saddr area for arguments
        movw    ax, sp
        movw    hl, ax
        mov     a, [hl + 6]             ; Argument p2 (low-order) passed via a stack
                                          and received via a register
        xch     a, x
        mov     a, [hl + 7]             ; Argument p2 (high-order) passed via a stack
                                          and received via a register
        movw    hl, ax                  ; Allocate arguments to HL
          .
          .
          .
        pop     ax
        movw    _@KREG12, ax            ; Restore the saddr area for argument
        pop     hl                      ; Restores the register for argument
        ret
```

11.7.4 norec function call interface (normal model)

(1) Passing arguments

All arguments are allocated to **_@NRARGx** and **_@RTARG6** and **7**. On the function caller, arguments are passed via register **_@NRARGx**.

On the function definition side, arguments passed via registers are copied to registers, or to **_@RTARG6** and **7** (Refer to **APPENDIX A LIST OF LABELS FOR saddr AREA**).

(2) Location and order of storing arguments

- On the function definition side, all arguments are allocated to registers, **_@NRARGx**, **_@RTARG6** and **7**. Arguments are allocated to **_@NRARGx** only when **-QR** is specified.
- Arguments are allocated to **_@RTARG6** and **7** only when there are arguments in DE (Refer to **APPENDIX A LIST OF LABELS FOR saddr AREA**).
- If there are arguments that are not allocated to registers, **_@NRARGx**, **_@RTARG6** and **7**, an error will result.
- On the function caller, arguments are passed via registers and **_@NRARGx**.
- On the function definition side, arguments that are passed via registers are copied to registers or **_@RTARG6** and **7**. Even when the arguments are passed via registers, register copying is necessary since the registers on the function caller (passing side) are different to those in the function definition side (receiving side). If the arguments are passed via registers, the area where the arguments are passed becomes the area to which they are allocated.
- If arguments can no longer be passed via a register, they can be allocated to **_@NRARGx** and passed via there. In this case, passing is carried out with registers and **_@NRARGx** intermingled.

(Argument allocation sequence)

- Arguments allocated to **_@NRARGx** are allocated in the sequence of declaration.
- Arguments allocated to registers are allocated to registers, **_@RTARG6** and **7** according to the following rules.

(Registers to be used)

• When one argument is used in **char**, **int**, **short**, **enum**, or pointer type:	**AX** pass, **DE** receive
• When two or more arguments are used in **char**, **int**, **short**, **enum**, or pointer type:	**AX** and **DE** pass, **_@RTARG6**, **7** and **DE** receive

(Allocation sequence)

• **char**, **int**, **short**, **enum**, and pointer type:	In the sequence of **DE**, **_@RTARG6** and **7**

(3) Location and order of storing automatic variables

The automatic variables are allocated to registers and **_@NRARGx** as long as there are allocable registers and **_@NRARGx**. If there is no allocable register, 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 a register, **_@NRARGx** and **_@NRATxx**, an error occurs.
The save and restoration of registers to which automatic variables are allocated are performed on the function definition side.

(Allocation sequence)

- The order of allocating automatic variables to registers, **_@RTARG6** to **7** is the same as the order of allocating arguments.
- The automatic variables allocated to **_@NRARGx**, **_@NRATxx** are allocated in the order of declaration.

[Example]

In the normal model

(C source)

```
norec void func3 (char, int, char, int) ;
void main ()
{
        func3 (0x12, 0x34, 0x56, 0x78) ;
}
norec void func3 (char p1, int p2, char p3, int p4)
{
        int a ;
        a = p2 ;
}
```

(Output code)

When **-QR** is specified

```
    _main :
    ; line   4 : func3 (0x12, 0x34, 0x56, 0x78) ;
         movw    _@NRARG1, #078H   ; 120      ; Argument is passed via _@NRARG1
         mov     _@NRARG0, #056H   ; 86       ; Argument is passed via _@NRARG0
         movw    de, #034H         ; 52       ; Argument is passed via register DE
         mov     a, #012H          ; 18       ; Argument is passed via register A
         call    !_func3                      ; Function call
         ret
    ; line   6 : norec void func3 (char p1, int p2, char p3, int p4)
    ; line   7 : {
    _func3 :
         mov     _@RTARG6, a                  ; Allocates the argument p1 to _@RTARG6
    ; line   8 : int a ;
    ; line   9 : a = p2 ;
         movw    ax, de                       ; Argument p2
         movw    _@NRARG2, ax      ; a        ; Automatic variable a
         ret
```

11.7.5 Static model function call interface

(1) Passing arguments

- On the function caller, both the register arguments and the normal arguments are passed in the same way. There can be a maximum of three arguments, up to 6 bytes, and all arguments are passed via registers.
- On the function definition side, the arguments passed via a register are stored in the area to which they are allocated. Register arguments are copied to registers. Even when the arguments are passed via registers, register copying is necessary since the registers on the function caller (passing side) are different to those on the function definition side (receiving side).
- Ordinary functions are allocated to the function-specific area.

(2) Location and order of storing arguments

(a) Argument storage location

- There are two types of arguments: arguments to be allocated to registers and normal arguments.
- The arguments allocated to registers are arguments that have undergone a register declaration.
- 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.
  Register arguments are copied to a register. Even when the arguments are passed via registers, register copying is necessary since the registers on the function caller (passing side) are different to those on the function definition side (receiving side). Normal arguments are allocated to the function-specific area.
- Saving and restoring registers to which arguments/automatic variables are allocated is performed on the function definition side.
- The remaining arguments are allocated to the function-specific area.
- On the function caller, both register arguments and normal arguments are passed in the same way. There can be a maximum of three arguments, up to 6 bytes, and all arguments are passed via a register. Refer to **Table 11-20** for the area to which arguments are passed.

Table 11-20 Areas to Which Arguments Are Passed in the Static Model

Data Size	The First Argument	The Second Argument	The Third Argument
1-byte data[Note]	**A**	**B**	**H**
2-byte data[Note]	**AX**	**BC**	**HL**
4-byte data[Note]	Allocates to **AX** and **BC** and the remainder allocated to **H** or **HL**.		

Note Neither structures nor unions are included in 1- to 4- byte data.

(b) Argument allocation sequence

- Arguments allocated to the function-specific area are allocated sequentially from the last argument.
- Register arguments are allocated to register **DE** according to the following rules.

(Registers to be used)
DE

(Allocation sequence)
char type: sequence of **D**, **E**
int, **short**, **enum** type: **DE**

(3) Location and order of storing automatic variables

(a) Storage location of automatic variables

- There are two types of automatic variables: automatic variables to be allocated to registers and normal automatic variables.
- Automatic variables allocated to registers are register-declared automatic variables and automatic variables specified at **-QV** specification.
- Register variables are allocated after register arguments are allocated. For this reason, the allocation of register variables to registers is performed only when registers are superfluous after register argument allocation.
- The remaining automatic variables are allocated to the function-specific area.
- Saving and restoring registers to which arguments are allocated is performed on the function definition side.

(b) Automatic variable allocation sequence

- Automatic variables are allocated to register **DE** according to the following rules.

(Registers to be used)
DE

(Allocation sequence)
char type: sequence of **E**, **D**
int, **short**, **enum** type: **DE**

- The automatic variables that are allocated to the function-specific area are allocated in the sequence of declaration.

[EXAMPLE 1]

(C source)

```
void func4 (register int, char) ; void func(void);
void main ()
{
        func4 (0x1234, 0x56) ;
}
void func4 (register int p1, char p2)
{
        register char r ;
        int a ;
        r = p2 ;
        a = p1 ; func ();
}
```

(Output code)

```
@@DATA      DSEG      UNITP
L0005 :     DS        (1)                  ; Argument p2
L0006 :     DS        (1)                  ; Automatic variable r
L0007 :     DS        (2)                  ; Automatic variable a

; line   1 : void func4 (register int, char) ; void func(void);
; line   2 : void main ()
; line   3 : {

@@CODE      CSEG
_main :
; line   4 : func4 (0x1234, 0x56) ;
     mov     b, #056H       ; 86           ; Pass the second argument via register B
     movw    ax, #01234H    ; 4660         ; Pass the first argument via register AX
     call    !_func4                       ; Function call
; line   5 : }
     ret
; line   6 : void func4 (register int p1, char p2)
; line   7 : {
_func4 :
     push    de                            ; Saves register for register argument
     movw    de, ax                        ; Allocate a register argument p1 to DE
     movw    a, b
     mov     !L0005, a                     ; Copy argument p2 to L0005
; line   8 : register char r ;
; line   9 : int a ;
; line  10 : r = p2 ;
     mov     !L0006, a                     ; r    ; Automatic variable r
; line  11 : a = p1 ; func();
     movw    ax, de                               ; Register argument p1
     movw    !L0007, ax                    ; a    ; Automatic variable a
     call    !_func
; line  12 : }
     pop     de                            ; Returns the register for register argument
     ret
```

[EXAMPLE 2]

(C source)

```
void func5 (int, register char) ; void func(void);
void main ()
{
        func5 (0x1234, 0x56) ;
}
void func5 (int p1, register char p2)
{
     register char r ;
     int a ;
     r = p2 ;
     a = p1 ; func();
}
```

(Output code)

When **-NQ** is specified

```
@@DATA     DSEG    UNITP
L0005 :    DS      (2)
L0006 :    DS      (2)

; line   1 : void func5 (int, register char) ; void func(void);
; line   2 : void main ()
; line   3 : {

@@CODE       CSEG
_main :
; line   4 : func5 (0x1234, 0x56) ;
             mov     b, #056H       ; 86   ; Pass the second argument via register B
             movw    ax, #01234H    ; 4660 ; Pass the first argument via register AX
             call    !_func5               ; Function call
; line   5 : }
             ret
; line   6 : void func5 (int p1, register char p2)
; line   7 : {
_func5 :
             push    de                    ; Saves a register for register variables and
                                           ; register arguments.
             movw    !L0005, ax            ; Copy argument p1 to L0005
             mov     a, b
```

(Output code …continued)

```
            mov     d, a                  ; Allocates a register argument p2 to d.
; line   8 : register char r ;
; line   9 : int a ;
; line  10 : r = p2 ;
            mov     a, d                  ; Register argument p2
            mov     e, a                  ; Register variable r
; line  11 : a = p1 ; func();
            movw    ax, !L0005     ; p1   ; Argument p1
            movw    !L0006, ax     ; a    ; Automatic variable a
            call    !_func
; line  12 : }
            pop     de                    ; Restores the register for register arguments
            ret
```

11.7.6 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**).

EXAMPLE 1

(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)

When **-QR** option is specified

```
_main:
; line  4   :func0 (0x1234, 0x5678) ;
     movw   ax, #05678H     ; 22136
     push   ax                         ; Stack is passed via the argument
     movw   ax, #01234H     ; 4660     ; The first argument that is passed via a register
     call   !_func0                    ; Function call
                                       ; Stack is not corrected here
; line  5   :}
     ret
; line  6   :_ _pascal void func0 (register int p1, int p2)
; line  7   :{
_func0:
     push   hl
     xch    a, x
     xch    a,_@KREG12
     xch    a, x
     xch    a,_@KREG13                 ; Allocate a register argument p1 to _@KREG12
     push   ax                         ; Saves the saddr area for register arguments
     movw   ax,_@KREG00
     push   ax                         ; Saves the saddr area for register variable
     push   ax                         ; Reserves the area automatic variable a
     movw   ax, sp
     movw   hl, ax
; line  8   :register int r ;
; line  9   :int a;
; line  10 : r = p2;
     mov    a, [hl + 10]    ; p2       ; Stack transfer argument p2
     xch    a, x
     mov    a, [hl + 11]    ; p2
     movw   _@KREG00, ax    ; r        ; Assign to register variable _@KREG00
; line   11 :a = p1 ;
     movw   ax,_@KREG12     ; p1       ; Register argument _@KREG12
     mov    [hl + 1], a     ; a
     xch    a, x
     mov    [hl], a         ; a        ; Assign to automatic variable a
; line   12 :}
     pop    ax                         ; Releases the automatic variable a area
     pop    ax
     movw   _@KREG00, ax               ; Restore the saddr area for register variable
     pop    ax
     movw   _@KREG12, ax               ; Restore the saddr area for register argument
     pop    hl
     pop    de              ; Obtains the return address
     pop    ax              ; The stack consumed by arguments passed via a stack is corrected
     push   de              ; Reloads the return address
     ret
```

EXAMPLE 2

(C source)

```
_ _pascal noauto void func2 (int, int) ;
void main ()
{
        func2 (0x1234, 0x5678) ;
}
_ _pascal noauto void func2 (int p1, int p2)
{
        .
        .
        .
}
```

(Output code)

When **-QR** option is specified

```
_main:
; line   4 : func2 (0x1234, 0x5678) ;
             movw   ax, #05678H ; 22136
             push   ax                     ; Argument passed via a stack
             movw   ax, #01234H ; 4660     ; The first argument that is passed via a register
             call   !_func2                ; Function call
                                           ; The stack is not corrected here
; line   5 : }
             ret
; line   6 : _ _pascal noauto void func2 (int p1, int p2)
; line   7 : {
_func2:
             push   hl                     ; Saves the register for arguments
             xch    a, x
             xch    a,_@KREG12             ; Allocate argument p1 to _@KREG12 (low-order)
             xch    a, x
             xch    a,_@KREG13             ; Allocate argument p1 to _@KREG13 (high-order)
             push   ax                     ; Saves the saddr area for arguments
             movw   ax, sp
             movw   hl, ax
             mov    a, [hl + 6]            ; Argument p2 (low-order) passed via a stack
                                           ; and received by a register
             xch    a, x
```

(Output code … continued)

```
        mov     a, [hl + 7]     ; Argument p2 (high-order) passed via a stack
                                ; and received by register
        movw    hl, ax          ; Allocates arguments to HL
              .
              .
              .
        pop     ax
        movw    _@KREG12, ax    ; Restore the saddr area for arguments
        pop     hl              ; Restores the register for arguments
        pop     de              ; Obtains the return address
        pop     ax              ; The stack consumed by arguments passed via a stack is corrected
        push    de              ; Reloads the return address
        ret
```

[MEMO]

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 RA78K0 Assembler Package and this C compiler is described in this order:

(1) Calling assembly language routines from the C language
(2) Calling C language functions from assembly language
(3) Referencing variables defined in the C language
(4) Referencing variables defined in assembly language on the C language side
(5) Cautions

12.1 Accessing Arguments/Automatic Variables

The procedure to access arguments and automatic variables of this C compiler is described in the following.

12.1.1 Normal model

- On the function call side, register arguments are passed in the same way as regular 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 data	AX	Stack passing
3-byte, 4-byte data	AX, BC	Stack passing
Floating-point number	AX, BC	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 to 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.
  Normal arguments passed via a register are pushed to 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 **HL** register.
  If the function argument is register declared or specified by the **-QV** option and specified by the **-QR** option, it is allocated to the **saddr** area.

Table 12-2 Storing of Arguments/Automatic Variables (Inside Called Function)

Option	Argument/auto Variable	Storage Location	Priority Level
-QV (register allocation option)	Declared argument or automatic variable	**HL** register (only when base pointer is not required)	**char** type: L, H, in this order **int, short, enum** type: HL
-QR (**saddr** allocation option)	register declared argument or automatic variable	**HL** register (only when base pointer is not required) Argument: **_@KREG12** to **15** [**0FEDCH** to **0FEDFH**] Automatic variable: **_@KREG00** to **11** [**0FED0H** to **0FEDBH**]	Only the number of bytes of the variable or argument is allocated, in order of appearance. Allocated to register as **char** type: L, H, in this order **int, short, enum** type: HL
-QRV	Declared argument or automatic variable	**HL** register (only when base pointer is not required) Argument: **_@KREG12** to **15** [**0FEDCH** to **0FEDFH**] Automatic variable: **_@KREG00** to **11** [**0FED0H** to **0FEDBH**]	Only the number of bytes of the variable or argument is allocated, in order of appearance. Allocated to register as **char** type: L, H, in this order **int, short, enum** type: HL
Default	Declared argument, automatic variable	Stack frame	Order of appearance

The following example shows the function call.

(**C source**: Normal model at the **-QRV** specification)

```
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)

```
    EXTRN       _@KREG12
    EXTRN       _@KREG13
    EXTRN       _@KREG00
    EXTRN       _@KREG02
    PUBLIC      _func0
    PUBLIC      _main

@@CODE          CSEG
_main:
    movw        ax,#05678H          ;22136
    push        ax                          ; Argument passed on stack
    movw        ax,#01234H          ;4660   ; 1st argument passed on register
    call        !_func0                     ; Function call
    pop         ax                          ; Argument passed on stack
    ret
_func0:
    push        hl                          ; Save the register for arguments
    xch         a,x
    xch         a,_@KREG12
    xch         a,x
    xch         a,_@KREG13                  ; Allocate register argument p1 to _@KREG12.
    push        ax                          ; Save the saddr area for register arguments.
    movw        ax,_@KREG00
    push        ax                          ; Save the saddr area for register variables.
    movw        ax,_@KREG02
    push        ax                          ; Save the saddr area for automatic variables.
    movw        ax,sp
    movw        hl,ax
    mov         a,[hl+10]                   ; Argument p2 passed on stack
    xch         a,x
    mov         a,[hl+11]
    movw        hl,ax                       ; Assigned to HL
    movw        ax,hl                       ; Argument p2
    movw        _@KREG00,ax         ;r      ; Assigned to register variables r.
    movw        ax,_@KREG12         ;p1     ; Register argument p1
    movw        _@KREG02,ax         ;a      ; Assigned to automatic variable a.
    pop         ax
    movw        _@KREG02,ax                 ; Restore the saddr area for register variables.
    pop         ax
    movw        _@KREG00,ax                 ; Restore the saddr area for automatic variables.
    pop         ax
    movw        _@KREG12,ax                 ; Restore the saddr area for register arguments.
    pop         hl                          ; Restore the register for arguments
    ret
    END
```

12.1.2 Static model

- On the function call side, register arguments are passed in the same way as regular arguments.
- Up to 3 arguments, or a total of 6 bytes, can be passed, all via a register.

Table 12-3 Passing Arguments (Function Call Side)

Type	Passing Location (First Argument)	Passing Location (Second Argument)	Passing Location (Third Argument)
1-byte data	A	B	H
2-byte data	AX	BC	HL
4-byte data	Allocated to AX and BC, remainder allocated to H or HL		

Remark 1- to 4-byte data does not include structures and unions.

- On the function definition side, arguments passed via a register are stored to the argument allocation location. Arguments (register arguments) declared with register are allocated to registers whenever possible, and regular arguments are allocated to areas reserved for specific functions.
- All register arguments are passed via registers, but the registers on the function call side (passing side) and the function definition side (receiving side) differ, and therefore register copying is performed.
- Saving and restoring of registers allocated an argument/automatic variable is performed on the function definition side.
- Function arguments and the values of automatic variables declared inside functions are stored to the function-specific areas listed below using an option. Function-specific areas are static areas in RAM reserved for each function.

Table 12-4 Storing of Arguments/Automatic Variables (Inside Called Function)

Option	Argument/auto Variable	Storage Location	Priority Level
-QV (**register** allocation option)	Declared argument or automatic variable	**DE** register	Arguments: **char** type: D, E, in this order **int, short, enum** type: DE Automatic variables: **char** type: E, D, in this order **int, short, enum** type: DE
Default	Declared argument, automatic variable	Function-specific area	Arguments are allocated starting from the 1st argument, automatic variables are allocated by order of appearance
Default	Argument, register variable declared with register	**DE** register	Only the number of bytes of the variable or argument is allocated, according to the number of times referenced. Other than the number of bytes of the variable or argument is allocated to the area peculiar to the function.

The following example shows the function call.

(**C source**: Static Model at **-SM** and **-QV** specifications)

```
    void sub();
    void func (register int, char);
    void main(){
              func (0x1234, 0x56);
    }
    void func (register int p1, char p2){
              register char r;
              int a;
              r=p2;
              a=p1;
    }         sub();
```

(Output assembler source)

```
    PUBLIC    _func
    PUBLIC    _main
              :
@@DATA        DSEG
?L0005:       DS    (1)                   ; Argument p2
?L0006:       DS    (1)                   ; Register variable r
?L0007:       DS    (2)                   ; Automatic variable a
              :
@@CODE        CSEG
_main:
    mov       b,#056H          ;86        ; Pass the 2nd argument by register B.
    movw      ax,#01234H       ;4660      ; Pass the 1st argument by register AX.
    call      !_func                      ; Function call
    ret

func:
    push      de                          ; Save registers for register arguments.
    movw      de,ax                       ; Allocate register arguments p1 to DE.
    mov       a,b
    mov       !?L0005,a                   ; Copy argument p2 to ?L0005.
    mov       !?L0006,a        ;r         ; Assigned to register variable r
    movw      ax,de                       ; Register argument p1
    movw      !?L0007,ax       ;a         ; Assigned to automatic variable a
    call      !_sub
    pop       de                          ; Restores the register for register arguments.
    ret
    END
```

12.2 Storing Return Values

Return values during function calls are stored to registers and carry flags.
The storage locations of return values are shown in the table below.

Table 12-5 Storage Location of Return Values

Type	Normal Model	Static Model
1-byte integer	BC	A
2-byte integer		AX
4-byte integer	BC (low-order), DE (high-order)	Not supported
Pointer	BC	AX
Structure, union	BC (start address of structure or union copied to function-specific area)	Not supported
1 bit	CY (carry flag)	CY (carry flag)
Floating-point number	BC (low-order), DE (high-order)	Not supported

12.3 Calling Assembly Language Routines from C Language

This section shows examples when the normal model (default) is used. If the **-QV** option, **-QR** option, and **-QRV** option are specified, arguments are stored as indicated in **Table 12-2**. However, the HL register is allocated only when no base pointer is required (when base pointer is not used).

Calling an assembly language routine from the C language is described as follows.

- C language function calling procedure
- Saving data from the assembly language routine and returning

(1) C language function calling procedure

This is a C language program example that calls an assembly language routine.

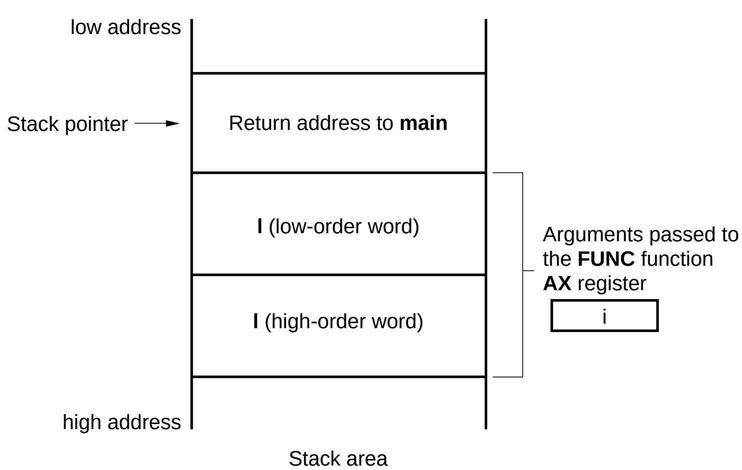
```
extern int FUNC(int, long);     /* Function prototype */

void main()
{
        int     i, j;
        long    l;

        i = 1;
        l = 0x54321;
        j = FUNC(i, l);         /* Function call */
}
```

In this program example, the interface and control flow with the program that is executing are as follows.

(1) Placing the first argument passed from the **main** function to the **FUNC** function in the register, and the second and subsequent arguments on the stack.
(2) Passing control to the **FUNC** function by using the CALL instruction.

The next figure shows the stack immediately after control moves to the **FUNC** function in the above program example.

Figure 12-1 Stack Area After a Call



(2) Saving data from the assembly language routine and returning

The following processing are performed in the **FUNC** function called from the **main** function.

(1) Save the base pointer, work register.
(2) Copy the stack pointer (**SP**) to the base pointer (**HL**).
(3) Perform the processing in the **FUNC** function.
(4) Set the return value.
(5) Restore the saved register.
(6) Return to the **main** function.

Next, an example of an assembly language program is explained.

```
$PROCESSOR(054)

    PUBLIC     _FUNC
    PUBLIC     _DT1
    PUBLIC     _DT2

@@DATA         DSEG      UNITP
_DT1:  DS      (2)
_DT2:  DS      (4)

@@CODE         CSEG
_FUNC:
    PUSH       HL                      ; save base pointer ---------------------------------(1)
    PUSH       AX
    MOVW       AX,SP                   ; copy stack pointer---------------------------------(2)
    MOVW       HL,AX
    MOV        A,[HL]                  ; arg1
    XCH        A,X
    MOV        A,[HL+1]                ; arg1
    MOVW       !_DT1,AX                ; move 1st argument(i)
    MOV        A,[HL+8]                ; arg2
    XCH        A,X
    MOV        A,[HL+9]                ; arg2
    MOVW       !_DT2+2,AX
    MOV        A,[HL+6]                ; arg2
    XCH        A,X
    MOV        A,[HL+7]                ; arg2
    MOVW       !_DT2,AX                ; move 2nd argument(l)
    MOVW       BC,#0AH                 ; set return value -----------------------------------(4)
    POP        AX
    POP        HL                      ; restore base pointer -------------------------------(5)
    RET ------------------------------------------------------------------------------------(6)
    END
```

(1) Saving base pointer, work register

A label with '_' prefixed to the function name described in the C source is described. Base pointers and work registers are saved with the same name as function names described inside the C source.

After the label is described, the HL register (base pointer) is saved.

In the case of programs generated by the C compiler, other functions are called without saving the register for register variables. Therefore, if changing the values of these registers for functions that are called, be sure to save the values beforehand. However, if register variables are not used on the call side, saving the work register is not required.

(2) Copying to base pointer (**HL**) of stack pointer (**SP**)

The stack pointer (**SP**) changes due to 'PUSH, POP' inside functions. Therefore, the stack pointer is copied to register 'HL' and used as the base pointer of arguments.

(3) Basic processing of **FUNC** function

After processings (1) and (2) are performed, the basic processing of called functions is performed.

(4) Setting the return value

If there is a return value, it is set in the 'BC' and 'DE' registers. If there is no return value, setting is unnecessary.

Return value of 16 or fewer bits :

BC register
word

Return value of 17 or more bits :

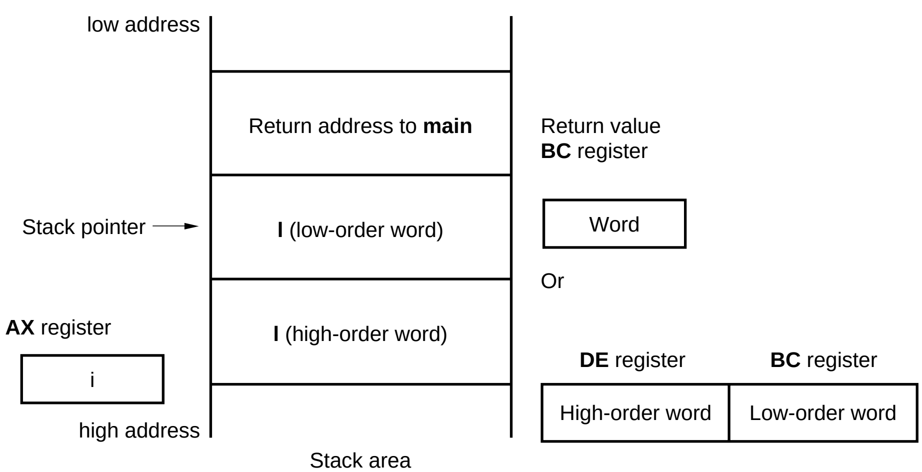
DE register	**BC** register
high-order word	low-order word

(5) Restoring the registers

Restore the saved base pointer and work register.

(6) Returning to the main function

Figure 12-2 Stack Area After Returning



12.4 Calling C Language Routines from Assembly Language

(1) Calling the C language function from an assembly language program

The procedure for calling a function written in the C language from an assembly language routine is:

(1) Place the arguments on the stack.
(2) Save the C work registers (AX, BC, and DE).
(3) Call the C language function.
(4) Increment the value of the stack pointer (**SP**) by the number of bytes of arguments.
(5) Reference the return value of the C language function (in **BC** or **DE** and **BC**).

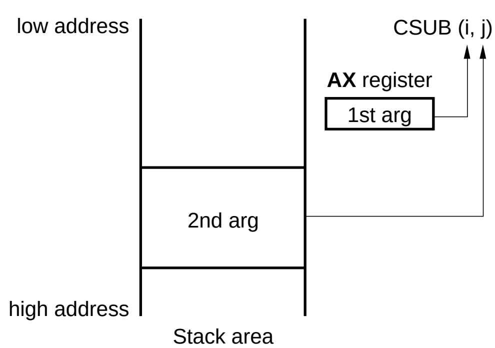
This is an example of an assembly language program.

```
$PROCESSOR (054)

      NAME   FUNC2
      EXTRN  _CSUB
      PUBLIC _FUNC2

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

(1) Stacking arguments

Any arguments are placed on the stack. **Figure 12-3** shows argument passing.

Figure 12-3 Placing Arguments on Stack



(2) Saving the work registers (AX, BC, and DE)
The three register pairs of AX, BC, and DE are used in the C language. Their values are not restored when returning. Therefore, if the values in registers are needed, they are saved on the calling side.
Save or restore the registers before or after an argument pass code. The HL register is always saved on the side of the C language when it is used in the C language.

(3) Calling a C language function
A CALL instruction calls a C language function. If the C language function is a **callt** function, the callt instruction performs the call, and if a **callf** function, the callf instruction performs it.

(4) Restoring the stack pointer (**SP**)
The stack pointer is restored by the number of bytes that hold the arguments.

(5) Referencing the return value (**BC** and **DE**)
The return value from the C language is returned as follows.

	BC register
Return value of 16 or fewer bits :	word

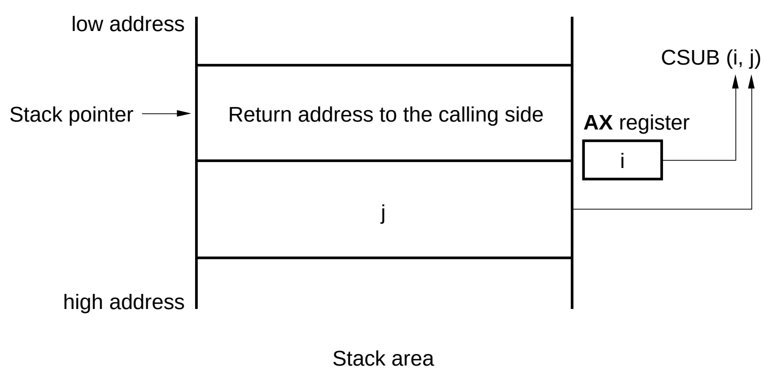
	DE register	**BC** register
Return value of 17 or more bits :	high-order word	low-order word

(2) Referencing arguments in a C language function

To correctly pass the i and j arguments to the C language program shown below, they are placed on the stack as shown in **Figure 12-4**.

```
void CSUB (i, j)
int      i, j ;
{
         i += j;
}
```

Figure 12-4 Passing Arguments to C Language



12.5 Referencing Variables Defined in Other Languages

(1) Referencing variables defined in the C language

If external variables defined in a C language program are referenced in an assembly language routine, the extern declaration is used. Underscores '_' are added to the beginning of the variables defined in the assembly language routine.

C language program example

```
     extern void subf();

     char   c = 0;
     int    i = 0;
     void main()
     {
            subf();
     }
```

The following occurs in the RA78K0 assembler.

```
$PROCESSOR (054)

     PUBLIC _subf
     EXTRN  _c
     EXTRN  _i

@@CODE CSEG
_subf:
     MOV    a, #04H
     MOV    !_c, a
     MOVW   ax, #07H     ;7
     MOVW   !_i,ax
     RET
     END
```

(2) Referencing variables defined in the assembly language from the C language

Variables defined in assembly language are referenced from the C language in this way.

C language program example

```
extern char c;
extern int i;

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

The following occurs in the RA78K0 assembler.

```
      NAME ASMSUB

      PUBLIC     _c
      PUBLIC     _i

ABC   CSEG
_c:   DB    0
_i:   DW    0

      END
```

12.6 Cautions

(1) '_' (underscore)

This C compiler adds an underscore '_' (ASCII code '5FH') to external definitions and reference names of the object modules to be output. In the next C program example, "j = FUNC(i, l);" is taken as a reference to the external name _FUNC.

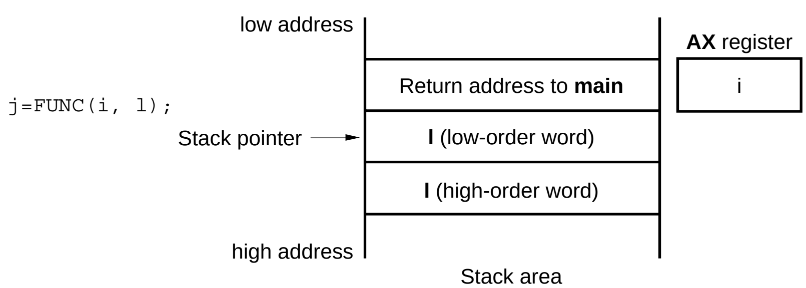
```
extern int FUNC(int, long);       /* Function prototype */

void main()
{
        int     i, j;
        long    l;

        i = 1;
        l = 0x54321;
        j = FUNC(i, l);           /* Function call */
}
```

The routine name is written as '_FUNC' in RA78K0.

(2) Argument positions on the stack

The arguments placed on the stack are placed from the postfix argument to the prefix argument in the direction from the high address to the low address.

Figure 12-5 Stack Positions of Arguments



[MEMO]

This chapter introduces how to effectively use this C compiler.

13.1 Efficient Coding

When developing 78K0 Series microcomputer-applied products, efficient object generation may be realized with this C compiler by utilizing the **saddr** area, **callt** table, or **callf** area of the device.

- Use external variables
  - if (**saddr** area is usable) —— **sreg/_ _sreg** variables are used/
    compiler option (**-RD**) is used

- Use 1-bit data
  - **if** (**saddr** is usable) —— **bit/boolean/ _ _boolean** type variables are used

- Function definition
  - **if** (function to be called several times)
    - **if** (**callt** area is usable)
      - Use as **_ _callt/callt** function (effective for reducing code size)
    - **if** (**callf** area is usable)
      - Use as **_ _callf/callf** function (effective for improving execution speed)
    - **if** (not used recursively)
      - Use as **_ _leaf/norec** function
  - **if** (automatic variables are not used)
    - Use as **noauto** function
  - **if** (automatic variables are used **&&saddr** area is usable)
    - **register** declaration

(1) Using external variable

When defining an external variable, specify the external variable to be defined as a **sreg/_ _sreg** variable if the **saddr** area can be used. Instructions to **sreg/_ _sreg** variables are shorter in code length than instructions to memory. This helps shorten object code and improve program execution speed. (The same can be also performed by specifying the **-RD** option, instead of using the **sreg** variable.)

```
Definition of sreg/_ _sreg variable:  extern sreg int variable-name ;
                                      extern_ _sreg int variable-name ;
```

Remark Refer to **11.5 (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 **bit/boolean/_ _boolean** type variable. Because saddr area is used as well as **sreg** variable, the codes can be shortened and the execution speed can be improved.

```
Declaration of bit/boolean type variable:  bit variable-name ;
                                           boolean variable-name ;
                                           _ _boolean variable-name ;
```

Remark Refer to **11.5 (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 call 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 **callt/callf** functions can be called faster than ordinary function calls with shorter codes because the **callt/callf** functions are called using the **callt/callf** area of the device.

```
Definition of callt function: callt int tsub() {
                                      .
                                      .
                                      .
                              }
Definition of callf function: callf int tsub() {
                                      .
                                      .
                                      .
                              }
```

Remark Refer to **11.5 (1) callt functions**, **11.5 (6) norec functions**, and **11.5 (13) callf functions**.

In addition to the use of the **saddr** area, the objects that do not need the modification of the C source by compiling with the optimization option can be generated. For the effect of each **-Q** suboption, refer to the **CC78K0 C Compiler Operation User's Manual**.

(4) Optimization option

The optimization options that emphasize the object code size the most is as follows.

[Object code is emphasized the most]

```
    -QX3
```

The further shortening of the code size and the improvement of the execution speed is possible by adding **_ _sreg** to variables. However, this is restricted to the cases when the **saddr** area can be used. When the areas are run out and cannot be used, a compile error occurs.
If execution speed is also highly emphasized, specify the **-QX2** default.
In addition, the object efficiency can be improved by adding the extended functions supported by this compiler to the C source.

(5) Using extended description

- Function definition
  - **if** (function to be called several times)
    - **if** (not used recursively)
      - Use as **_ _leaf/norec** function
  - **if** (automatic variables not used)
    - Use as **noauto** function
  - **if** (automatic variables are used **&& register/saddr** area is usable)
    - **register** declaration
  - **if** (internal **static** variables are used) && (**saddr** area is usable)
    - **_ _sreg** declaration

- 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 **norec** function (**norec int rout ()**...), refer to **11.5 (6) norec functions** and **11.7.4 norec function call interface**.

- Functions which 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 formation and their arguments will be passed to registers as much as possible. These functions help shorten object code and improve program execution speed.

  Remark Refer to **11.5 (5) noauto functions, 11.7.3 noauto function call interface** about **noauto** function definition **(noauto int sub1 (int i) ...)**.

- Functions which use automatic variables
  If the **saddr** area can be used for a function that does not use automatic variables, declare the function with the **register** storage class specifier. By this **register** declaration, the object declared as register will be allocated to a register. A program using registers operates faster than that using memory and object code can be shortened as well.

  Remark Refer to **11.5 (2) Register variables** about definition of **register** variable **(register int i; ...)**.

- Functions which use internal static variables
  If the **saddr** area can be used for a function that uses internal static variables, declare the function with **_ _sreg** or specify the **-RS** option. In the same way as with **sreg** variables, the object code can be shortened and the execution speed can be improved.

  Remark Refer to **11.5 (3) How to use the saddr area**.

In addition, the code efficiency and the execution speed can be improved in the following method.

- Use of SFR name (or SFR bit name).

```
#pragma sfr
```

- Use of **_ _sreg** declaration for bit fields which consist only of 1-bit members (unsigned char type can be used for members).

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

- Use of the register bank change for interrupt processing.

```
#pragma interrupt INTP0 inter RB1
```

- Use of multiplication and division embedded function.

```
#pragma mul
#pragma div
```

- Description of only the modules whose speed needs to be improved in the assembly language.

APPENDIX A LIST OF LABELS FOR saddr AREA

In the CC78K0, the **saddr** area is referenced by the following label names. Therefore, the label names in the C source program and in assembler source program that have the same names as the following cannot be used.

A.1 Normal Model

(a) Register variables

Label Name	Address
_@KREG00	0FED0H
_@KREG01	0FED1H
_@KREG02	0FED2H
_@KREG03	0FED3H
_@KREG04	0FED4H
_@KREG05	0FED5H
_@KREG06	0FED6H
_@KREG07	0FED7H
_@KREG08	0FED8H
_@KREG09	0FED9H
_@KREG10	0FEDAH
_@KREG11	0FEDBH
_@KREG12	0FEDCH[Note]
_@KREG13	0FEDDH[Note]
_@KREG14	0FEDEH[Note]
_@KREG15	0FEDFH[Note]

Note When the arguments of the function are declared by register or the **-QV** option is specified and the **-QR** option is specified, arguments are allocated to the **saddr** area.

(b) Arguments of norec function

Label Name	Address
_@NRARG0	0FEC0H
_@NRARG1	0FEC2H
_@NRARG2	0FEC4H
_@NRARG3	0FEC6H

(c) Automatic variables of norec function

Label Name	Address
_@NRAT00	0FEC8H
_@NRAT01	0FEC9H
_@NRAT02	0FECAH
_@NRAT03	0FECBH
_@NRAT04	0FECCH
_@NRAT05	0FECDH
_@NRAT06	0FECEH
_@NRAT07	0FECFH

(d) Arguments of runtime library

Label Name	Address
_@RTARG0	0FEB8H
_@RTARG1	0FEB9H
_@RTARG2	0FEBAH
_@RTARG3	0FEBBH
_@RTARG4	0FEBCH
_@RTARG5	0FEBDH
_@RTARG6	0FEBEH
_@RTARG7	0FEBFH

A.2 Static Model

(a) Shared area

Label Name	Address
_@KREG00	0FED0H
_@KREG01	0FED1H
_@KREG02	0FED2H
_@KREG03	0FED3H
_@KREG04	0FED4H
_@KREG05	0FED5H
_@KREG06	0FED6H
_@KREG07	0FED7H
_@KREG08	0FED8H
_@KREG09	0FED9H
_@KREG10	0FEDAH
_@KREG11	0FEDBH
_@KREG12	0FEDCH
_@KREG13	0FEDDH
_@KREG14	0FEDEH
_@KREG15	0FEDFH

(b) For arguments, automatic variables, and work

Label Name	Address
_@NRAT00	0FExxH[Note]
_@NRAT01	_@NRAT00 + 1
_@NRAT02	_@NRAT00 + 2
_@NRAT03	_@NRAT00 + 3
_@NRAT04	_@NRAT00 + 4
_@NRAT05	_@NRAT00 + 5
_@NRAT06	_@NRAT00 + 6
_@NRAT07	_@NRAT00 + 7

Note Arbitrary address in the **saddr** area

[MEMO]

APPENDIX B LIST OF SEGMENT NAMES

This chapter explains all the segments that the compiler outputs and their locations.
(1) and (2) show the option and re-allocation attributes used in the table.

This section describes all the segments and allocations that are output by compiler.

<1> CSEG re-allocation attribute

CALLT0: Allocates the specified segment so that the start address is a multiple of two within the range of 40H to 7FH.

AT absolute expression: Allocates the specified segment to an absolute address (within the range of 0000H to FEFFH).

FIXED: Allocates the start address of the specified segment within the range of 800H to 0FFFH.

UNITP: Allocates the specified segment so that the start address is a multiple of two within any position (within the range of 80H to 0FA7EH).

<2> DSEG re-allocation attribute

SADDRP: Allocates the specified segment so that the start address is a multiple of two within the range of FE20H to FEFFH in the saddr area.

UNITP: Allocates the specified segment so that the start address is a multiple of two within any position (default is within the RAM area).

B.1 List of Segment Names

B.1.1 Program area and data area

Section Name	Segment Type	Re-allocation Attribute	Description
@@CODE	CSEG		Segment for code portion
@@CNST	CSEG	UNITP	Segment for **const** variable
@@R_INIT	CSEG	UNITP	Segment for initialization data (with initial value)
@@R_INIS	CSEG	UNITP	Segment for initialization data (**sreg** variable with initial value)
@@CALF	CSEG	FIXED	Segment for **callf** function
@@CALT	CSEG	CALLT0	Segment for **callt** function table
@@VECTnn	CSEG	AT 00nnH	Segment for vector table[Note]
@@INIT	DSEG	UNITP	Segment for data area (with initial value)
@@DATA	DSEG	UNITP	Segment for data area (without initial value)
@@INIS	DSEG	SADDRP	Segment for data area (**sreg** variable with initial value)
@@DATS	DSEG	SADDRP	Segment for data area (**sreg** variable without initial value)
@@BITS	BSEG		Segment for **boolean**-type and **bit**-type variables
@@BANK1 to @@BANK15	CSEG		Segment for bank function

Note The value of nn changes depending on the interrupt types.

B.1.2 Flash memory area

Section Name	Segment Type	Re-allocation Attribute	Description
@ECODE	CSEG		Segment for code portion
@ECNST	CSEG	UNITP	Segment for **const** variable
@ER_INIT	CSEG	UNITP	Segment for initialization data (with initial value)
@ER_INIS	CSEG	UNITP	Segment for initialization data (**sreg** variable with initial value)
@ECALF	CSEG	FIXED	Segment for **callf** function
@ECALT	CSEG	CALLT0	Segment for **callt** function table
@EVECTnn	CSEG	AT 20mmH	Segment for vector table[Note1]
@EXTxx	CSEG	AT 206yH	Segment for flash area branch table (when **-ZF** is specified)[Note2]
@EINIT	DSEG	UNITP	Segment for data area (with initial value)
@EDATA	DSEG	UNITP	Segment for data area (without initial value)
@EINIS	DSEG	SADDRP	Segment for data area (**sreg** variable with initial value)
@EDATS	DSEG	SADDRP	Segment for data area (**sreg** variable without initial value)
@EBITS	BSEG		Segment for **boolean**-type and **bit**-type variables
@EBANK1 to @EBANK15	CSEG		Segment for bank function

Note1 The value of nn and mm changes depending on the interrupt types.

2 The values of xx and y vary depending on the ID of the flash area function.

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 **RA78K0 User's Manual** for details.

```
; 78K/0 Series C Compiler V3.30 Assembler Source
;                                           Date : xx xxx xxxx   Time : xx : xx :
xx

; Command    : -c014 sample.c -sa -ng
; In-file    : sample.c
; Asm-file   : sample.asm
; Para-file  :

$PROCESSOR (014)
$NODEBUG
$NODEBUGA
$KANJICODE SJIS
$TOL_INF            03FH, 0330H, 00H, 020H, 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                        ; Segment for boolean-type variable
_b_bit      DBIT

@@CNST      CSEG    UNITP               ; Segment for const variable
_i_cnst:    DW      01H         ; 1

@@R_INIT    CSEG    UNITP               ; Segment for initialization data
                                          (External variable with initial value)
            DW      00002H, 00000H ; 2

@@INIT      DSEG    UNITP               ; Segment for data area
                                          (External variable with initial value)
_l_init:    DS      (4)

@@DATA      DSEG    UNITP               ; Segment for data area
```

```
                                                 (External variable without initial value)
_i_data:    DS      (2)

@@R_INIS    CSEG    UNITP                       ; Segment for initialization data
                                                 (sreg variable with initial value)
            DW      03H     ; 3

@@INIS      DSEG    SADDRP                      ; Segment for data area
                                                 (sreg variable with initial value)
_sr_inis:   DS      (2)

@@DATS      DSEG    SADDRP                      ; Segment for data area
                                                 (sreg variable without initial value)
_sr_dats:   DS      (2)

@@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 hl                                   ; [INF] 1, 4
; line  15 :  int i ;
; line  16 :  i = 100 ;
       movw  hl, #064H    ; 100                   ; [INF] 3, 6
; line  17 :  }
        pop  hl                                   ; [INF] 1, 4
        ret                                       ; [INF] 1, 6
; line  18 :
; line  19 :  void inter ()                      /* Interrupt function definition */
; line  20 :  {
_inter:
        sel  RB1                                  ; [INF] 2, 4  Selects register bank 1
        push hl                                   ; [INF] 1, 4
```

```
; line  21 :  unsigned char uc = 0;
        mov  l, #00H ; 0                            ; [INF] 2, 4
; line  22 :  uc++ ;
        inc  l                                      ; [INF] 1, 2
; line  23 : if (b_bit)
        bf   _b_bit, $L0005                         ; [INF] 4, 10
; line  24 : b_bit = 0 ;
        clr1 _b_bit                                 ; [INF] 2, 4
L0005:
; line  25 :  }
        pop  hl                                     ; [INF] 1, 4
        reti                                        ; [INF] 1, 6
; line  26 :
; line  27 :  callt void f_clt ()                 /* callt function definition */
; line  28 :  {
_f_clt:
; line  29 :  }
        ret                                         ; [INF] 1, 6
; 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                                         ; [INF] 1, 6
@@VECT06     CSEG    AT      0006H        ; Interrupt vector
_@vect06 :
        DW   _inter
        END

; *** Code Information ***
;
; $FILE C: /NECTools32/work/sample.c
;
; $FUNC main(14)
;       void=(void)
;       CODE SIZE= 6 bytes, CLOCK_SIZE= 20 clocks, STACK_SIZE= 2 bytes
;
; $FUNC inter(20)
;       void=(void)
;       CODE SIZE= 14 bytes, CLOCK_SIZE= 38 clocks, STACK_SIZE= 2 bytes
;
; $FUNC f_clt(28)
;       void=(void)
;       CODE SIZE= 1 bytes, CLOCK_SIZE= 6 clocks, STACK_SIZE= 0 bytes
;
; $FUNC f_clf(32)
```

```
;       void=(void)
;       CODE SIZE= 1 bytes, CLOCK_SIZE= 6 clocks, STACK_SIZE= 0 bytes

; Target chip: uPD78014
; Device file: Vx.xx
```

[MEMO]

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**, **cstarte**, **cprep**, and **cdisp** are called in the format with _@ attached to the top.

No library supports is available for operations not in **Table C-1**. The compiler executes in-line development.

long addition and subtraction, **and/or/xor** and shift may be developed in-line.

Table C-1 List of Runtime Libraries (1/5)

Classification	Function Name	Supported Model		Function
		Normal Model	Static Model	
Increment	lsinc	√	–	Increments **signed long**
	luinc	√	–	Increments **unsigned long**
	finc	√	–	Increments **float**
Decrement	lsdec	√	–	Decrements **signed long**
	ludec	√	–	Decrements **unsigned long**
	fdec	√	–	Decrements **float**
Sign reverse	lsrev	√	–	Reverses the sign of **signed long**
	lurev	√	–	Reverses the sign of **unsigned long**
	frev	√	–	Reverses the sign of **float**
1's complement	lscom	√	–	Obtains 1's complement of **signed long**
	lucom	√	–	Obtains 1's complement of **unsigned long**
Logical NOT	lsnot	√	–	Negates **signed long**
	lunot	√	–	Negates **unsigned long**
	fnot	√	–	Negates **float**
Multiply	csmul	√	√	Performs multiplication between **signed char** data
	cumul	√	√	Performs multiplication between **unsigned char** data
	ismul	√	√	Performs multiplication between **signed int** data
	iumul	√	√	Performs multiplication between **unsigned int** data
	lsmul	√	–	Performs multiplication between **signed long** data
	lumul	√	–	Performs multiplication between **unsigned long** data
	fmul	√	–	Performs multiplication between **float** data
Divide	csdiv	√	√	Performs division between **signed char** data
	cudiv	√	√	Performs division between **unsigned char** data
	isdiv	√	√	Performs division between **signed int** data
	iudiv	√	√	Performs division between **unsigned int** data
	lsdiv	√	–	Performs division between **signed long** data
	ludiv	√	–	Performs division between **unsigned long** data
	fdiv	√	–	Performs division between **float** data

Table C-1 List of Runtime Libraries (2/5)

Classification	Function Name	Supported Model		Function
		Normal Model	Static Model	
Remainder	csrem	√	√	Obtains remainder after division between **signed char** data
	curem	√	√	Obtains remainder after division between **unsigned char** data
	isrem	√	√	Obtains remainder after division between **signed int** data
	iurem	√	√	Obtains remainder after division between **unsigned int** data
	lsrem	√	–	Obtains remainder after division between **signed long** data
	lurem	√	–	Obtains remainder after division between **unsigned long** data
Add	lsadd	√	–	Performs addition between **signed long** data
	luadd	√	–	Performs addition between **unsigned long** data
	fadd	√	–	Performs addition between **float** data
Subtract	lssub	√	–	Performs subtraction between **signed long** data
	lusub	√	–	Performs subtraction between **unsigned long** data
	fsub	√	–	Performs subtraction between **float** data
Shift left	lslsh	√	–	Shifts **singed long** data to the left
	lulsh	√	–	Shifts **unsigned long** data to the left
Shift right	lsrsh	√	–	Shifts **signed long** data to the right
	lursh	√	–	Shifts **unsigned long** data to the right
Compare	cscmp	√	√	Compares **signed char** data
	iscmp	√	√	Compares **signed int** data
	lscmp	√	–	Compares **signed long** data
	lucmp	√	–	Compares **unsigned long** data
	fcmp	√	–	Compares **float** data
Bit AND	lsband	√	–	Performs an AND operation between **signed long** data
	luband	√	–	Performs an AND operation between **unsigned long** data
Bit OR	lsbor	√	–	Performs an OR operation between **signed long** data
	lubor	√	–	Performs an OR operation between **unsigned long** data
Bit XOR	lsbxor	√	–	Performs an XOR operation between **signed long** data
	lubxor	√	–	Performs an XOR operation between **unsigned long** data
Logical AND	fand	√	–	Performs a logical AND operation between two **float** data
Logical OR	for	√	–	Performs a 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**
Conversion from **bit**	btol	√	–	Converts from **bit** to **long**

Table C-1 List of Runtime Libraries (3/5)

Classification	Function Name	Supported Model		Function
		Normal Model	Static Model	
Startup routine	cstart	√	√	Startup module • After an area (2 × 32 bytes) where a function that will be registered is reserved with the **atexit** function, sets the beginning label name to **_@FNCTBL**. • Reserve a break area (32 bytes), sets the beginning label name to **_@MEMTOP**, and then sets the next label name of the area to **_@MEMBTM**. • Define the segment in the reset vector table as follows, and set the beginning address of the startup module. **@@VECT00 CSEG AT 0000H** **DW _@cstart** • Set the register bank to **RB0**. • Set 0 to the variable **_errno** to which the error number is input. • Set the variable **_@FNCENT**, to which the number of functions registered by the **atexit** function is input, to 0. • Set the address of **_@MEMTOP** to the variable **_@BRKADR** as the initial break value. • Set 1 as the initial value for the variable **_@SEED**, which is the source of pseudo random numbers for the **rand** function. • Perform copy processing of initialized data and execute 0 clear of external data without an initial value. • Call the **main** function (user program) • Call the **exit** function by parameter 0.
Pre- and post-processing of function	cprep	√	–	Pre-processing of function
	cdisp	√	–	Post-processing of function
	cprep2	√	–	Pre-processing of function (including the **saddr** area for register variables)
	cdisp2	√	–	Post-processing of function (including the **saddr** area for register variables)
	nrcp2	–	√	For copying arguments
	nrcp3	–	√	
	krcp2	–	√	
	krcp3	–	√	
	nkrc3	–	√	
	nrip2	–	√	
	nrip3	–	√	
	krip2	–	√	
	krip3	–	√	
	nkri31	–	√	
	nkri32	–	√	
	nrsave	–	√	For saving **_@NRATxx**
	nrload	–	√	For restoring **_@NRATxx**

Table C-1 List of Runtime Libraries (4/5)

Classification	Function Name	Supported Model		Function
		Normal Model	Static Model	
Pre- and post-processing of function	krs02	–	√	For saving **_@KREGxx**
	krs04	–	√	
	krs04i	–	√	
	krs06	–	√	
	krs06i	–	√	
	krs08	–	√	
	krs08i	–	√	
	krs10	–	√	
	krs10i	–	√	
	krs12	–	√	
	krs12i	–	√	
	krs14	–	√	
	krs14i	–	√	
	krs16	–	√	
	krs16i	–	√	
	krl02	–	√	For restoring **_@KREGxx**
	krl04	–	√	
	krl04i	–	√	
	krl06	–	√	
	krl06i	–	√	
	krl08	–	√	
	krl08i	–	√	
	krl10	–	√	
	krl10i	–	√	
	krl12	–	√	
	krl12i	–	√	
	krl14	–	√	
	krl14i	–	√	
	krl16	–	√	
	krl16i	–	√	
	hdwinit	√	√	Performs initialization processing of peripheral devices (sfr) immediately after CPU reset.
Bank function	fcall	√	–	Calls the bank function
BCD-type conversion	bcdtob	√	√	Converts 1-byte **bcd** to 1-byte **binary**
	btobcd	√	√	Converts 1-byte **binary** to 2-byte **bcd**
	bcdtow	√	√	Converts 2-byte **bcd** to 2-byte **binary**
	wtobcd	√	√	Converts 2-byte **binary** to 2-byte **bcd**
	bbcd	√	√	Converts 1-byte **binary** to 1-byte **bcd**
Auxiliary	mulu	√	√	**mulu** instruction-compatible
	mulue	√	√	**mulu** instruction-compatible
	divuw	√	√	**divuw** instruction-compatible
	divuwe	√	√	**divuw** instruction-compatible

Table C-1 List of Runtime Libraries (5/5)

Classification	Function Name	Supported Model		Function
		Normal Model	Static Model	
Auxiliary	addwbc	√	√	For replacing the fixed-type instruction pattern
	clra0	√	√	
	clra1	√	√	
	clrx0	√	√	
	clrax0	√	√	
	clrax1	–	√	
	clrbc0	√	–	
	clrbc1	√	–	
	cmpa0	√	√	
	cmpa1	√	√	
	cmpc0	√	–	
	cmpax1	√	√	
	ctoi	√	√	
	maxde	√	√	
	mdeax	√	√	
	incde	√	√	
	decde	√	√	
	maxhl	√	√	
	mhlax	√	√	
	inchl	√	√	
	dechl	√	√	
	dellab	√	–	
	dell03	√	–	
	della4	√	–	
	delsab	√	–	
	dels03	√	–	
	hlllab	√	–	
	hlll03	√	–	
	hllla4	√	–	
	hllsab	√	–	
	hlls03	√	–	
	apinch	√	√	
	apdech	√	√	
	incwhl	√	√	
	decwhl	√	√	
	shl4	√	√	
	shr4	√	√	
	swap4	√	√	
	tableh	√	√	
	uctoi	√	√	

[MEMO]

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	Normal Model	Static Model
ctype.h	isalnum	0	0
	isalpha	0	0
	iscntrl	0	0
	isdigit	0	0
	isgraph	0	0
	islower	0	0
	isprint	0	0
	ispunct	0	0
	isspace	0	0
	isupper	0	0
	isxdigit	0	0
	tolower	0	0
	toupper	0	0
	isascii	0	0
	toascii	0	0
	_tolower	0	0
	_toupper	0	0
	tolow	0	0
	toup	0	0
setjmp.h	setjmp	4	4
	longjmp	2	2
stdarg.h	va_arg	0	—
	va_start	0	—
	va_end	0	—
stdio.h	sprintf	52 (72)[Note 1]	—
	sscanf	290 (304)[Note 1]	—
	printf	54 (72)[Note 1]	—
	scanf	294 (304)[Note 1]	—
	vprintf	52 (72)[Note 1]	—
	vsprintf	52 (72)[Note 1]	—
	getchar	0	0
	gets	6	6
	putchar	0	0
	puts	4	4
stdlib.h	atoi	4	2
	atol	10	—
	strtol	18	—

Table D-1 List of Standard Library Stack Consumption (2/4)

Classification	Function Name	Normal Model	Static Model
stdlib.h	strtoul	18	—
	calloc	14	14
	free	8	8
	malloc	6	6
	realloc	10	12
	abort	0	0
	atexit	0	0
	exit	2 + n[Note 2]	2 + n[Note 2]
	abs	0	0
	div	6	—
	labs	2	—
	ldiv	14	—
	brk	0	0
	sbrk	4	4
	atof	35	—
	strtod	35	—
	itoa	10	10
	ltoa	16	—
	ultoa	16	—
	rand	14	—
	srand	0	—
	bsearch	32 + n[Note 3]	—
	qsort	16 + n[Note 4]	—
	strbrk	0	0
	strsbrk	4	4
	stritoa	10	10
	strltoa	16	—
	strultoa	16	—
string.h	memcpy	4	6
	memmove	4	6
	strcpy	2	4
	strncpy	4	6
	strcat	2	4
	strncat	4	6
	memcmp	2	4
	strcmp	2	2
	strncmp	2	4
	memchr	2	2
	strchr	4	0
	strcspn	6	6
	strpbrk	4	4

Table D-1 List of Standard Library Stack Consumption (3/4)

Classification	Function Name	Normal Model	Static Model
string.h	strrchr	4	4
	strspn	6	6
	strstr	4	4
	strtok	4	4
	memset	4	4
	strerror	0	0
	strlen	0	0
	strcoll	2	2
	strxfrm	4	4
math.h	acos	22	—
	asin	22	—
	atan	22	—
	atan2	23	—
	cos	24 (34)[Note 5]	—
	sin	24 (34)[Note 5]	—
	tan	28 (34)[Note 5]	—
	cosh	24	—
	sinh	27	—
	tanh	32	—
	exp	24	—
	frexp	2 (10)[Note 5]	—
	ldexp	2 (10)[Note 5]	—
	log	24 (34)[Note 5]	—
	log10	22 (32)[Note 5]	—
	modf	2 (10)[Note 5]	—
	pow	26 (36)[Note 5]	—
	sqrt	16	—
	ceil	2 (10)[Note 5]	—
	fabs	0	—
	floor	2 (10)[Note 5]	—
	fmod	2 (10)[Note 5]	—
	matherr	0	—
	acosf	22	—
	asinf	22	—
	atanf	22	—
	atan2f	23	—
	cosf	24 (34)[Note 5]	—
	sinf	24 (34)[Note 5]	—
	tanf	28 (34)[Note 5]	—
	coshf	24	—
	sinhf	27	—

Table D-1 List of Standard Library Stack Consumption (4/4)

Classification	Function Name	Normal Model	Static Model
math.h	tanhf	32	—
	expf	24	—
	frexpf	2 (10)[Note 5]	—
	ldexpf	2 (10)[Note 5]	—
	logf	24 (34)[Note 5]	—
	log10f	22 (32)[Note 5]	—
	modff	2 (10)[Note 5]	—
	powf	26 (36)[Note 5]	—
	sqrtf	16	—
	ceilf	2 (10)[Note 5]	—
	fabsf	0	—
	floorf	2 (10)[Note 5]	—
	fmodf	2 (10)[Note 5]	—
assert.h	_ _assertfail	64 (82)[Note 6]	—

Notes
1. Values in parentheses are for when the version that supports floating-point numbers is used.
2. n is the total stack consumption among external functions registered by the **atexit** function.
3. n is the stack consumption of external functions called from **bsearch**.
4. n is (20 + stack consumption of external functions called from **qsort**) × (1 + number of times recursive calls occurred).
5. Values in parentheses are for when an operation exception occurs.
6. 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 (1/4)

Classification	Function Name	Normal Model	Static Model
Increment	lsinc	0	—
	luinc	0	—
	finc	16 (26)Note 1	—
Decrement	lsdec	0	—
	ludec	0	—
	fdec	16 (26)Note 1	—
Sign reverse	lsrev	0	—
	lurev	0	—
	frev	0	—
1's complement	lscom	0	—
	lucom	0	—
Logical NOT	lsnot	0	—
	lunot	0	—
	fnot	0	—
Multiply	csmul	2	2
	cumul	2	2
	ismul	6	6
	iumul	6	6
	lsmul	6	—
	lumul	6	—
	fmul	10 (20)Note 1	—
Divide	csdiv	8	8
	cudiv	2	2
	isdiv	10	12
	iudiv	6	6
	lsdiv	10	—
	ludiv	6	—
	fdiv	10 (20)Note 1	—
Remainder	csrem	8	10
	curem	2	4
	isrem	10	12
	iurem	6	6
	lsrem	10	—
	lurem	6	—
Add	lsadd	0	—
	luadd	0	—
	fadd	10 (20)Note 1	—
Subtract	lssub	0	—
	lusub	0	—
	fsub	10 (20)Note 1	—

Table D-2 List of Runtime Library Stack Consumption (2/4)

Classification	Function Name	Normal Model	Static Model
Shift left	lslsh	2	—
	lulsh	2	—
Shift right	lsrsh	2	—
	lursh	2	—
Compare	cscmp	0	2
	iscmp	2	2
	lscmp	2	—
	lucmp	2	—
	fcmp	4 (16)Note 1	—
Bit AND	lsband	0	—
	luband	0	—
Bit OR	lsbor	0	—
	lubor	0	—
Bit XOR	lsbxor	0	—
	lubxor	0	—
Logical AND	fand	0	—
Logical OR	for	0	—
Conversion from floating-point number	ftols	8	—
	ftolu	8	—
Conversion to floating-point number	lstof	12 (22)Note 1	—
	lutof	12 (22)Note 1	—
Conversion from **bit**	btol	0	—
Startup routine	cstart	2	2
Pre- and post-processing of function	cprep	2 + nNote 2	—
	cdisp	0	—
	cprep2	Size of automatic variable + register variable	—
	cdisp2	0	—
	nrcp2	—	0
	nrcp3	—	0
	krcp2	—	0
	krcp3	—	0
	nkrc3	—	0
	nrip2	—	0
	nrip3	—	0
	krip2	—	0
	krip3	—	0
	nkri31	—	0
	nkri32	—	0
	nrsave	—	8
	nrload	—	0

Table D-2 List of Runtime Library Stack Consumption (3/4)

Classification	Function Name	Normal Model	Static Model
Pre- and post-processing of function	krs02	—	2
	krs04	—	4
	krs04i	—	4
	krs06	—	6
	krs06i	—	6
	krs08	—	8
	krs08i	—	8
	krs10	—	10
	krs10i	—	10
	krs12	—	12
	krs12i	—	12
	krs14	—	14
	krs14i	—	14
	krs16	—	16
	krs16i	—	16
	krl02	—	0
	krl04	—	0
	krl04i	—	0
	krl06	—	0
	krl06i	—	0
	krl08	—	0
	krl08i	—	0
	krl10	—	0
	krl10i	—	0
	krl12	—	0
	krl12i	—	0
	krl14	—	0
	krl14i	—	0
	krl16	—	0
	krl16i	—	0
	hdwinit	0	0
Bank function	fcall	6	—
BCD-type conversion	bcdtob	4	4
	btobcd	4	4
	bcdtow	4	4
	wtobcd	6	6
	bbcd	4	4
Auxiliary	mulu	4	4
	mulue	4	4
	divuw	6	6
	divuwe	6	6
	addwbc	0	0

Table D-2 List of Runtime Library Stack Consumption (4/4)

Classification	Function Name	Normal Model	Static Model
Auxiliary	clra0	0	0
	clra1	0	0
	clrx0	0	0
	clrax0	0	0
	clrax1	—	0
	clrbc0	0	—
	clrbc1	0	—
	cmpa0	0	0
	cmpa1	0	0
	cmpc0	0	—
	cmpax1	0	0
	ctoi	0	0
	maxde	0	0
	mdeax	0	0
	incde	0	0
	decde	0	0
	maxhl	0	0
	mhlax	0	0
	inchl	0	0
	dechl	0	0
	dellab	0	—
	dell03	0	—
	della4	0	—
	delsab	0	—
	dels03	0	—
	hlllab	0	—
	hlll03	0	—
	hllla4	0	—
	hllsab	0	—
	hlls03	0	—
	apinch	0	0
	apdech	0	0
	incwhl	0	0
	decwhl	0	0
	shl4	0	0
	shr4	0	0
	swap4	0	0
	tableh	0	0
	uctoi	0	0

Notes 1. Values in parentheses are for when an operation exception occurs (when the **matherr** function included with the compiler is used).

2. n is the size of the automatic variable to be secured.

[MEMO]