NEC

USER'S MANUAL

CC78K0S C COMPILER

LANGUAGE

© NEC Corporation 1996

Document No. U11817EJ1V0UM00 (1st edition)
Date Published October 1996 P
Printed in Japan

MS-DOS™ is a trademark of Microsoft Corporation.

The export of this product from Japan is prohibited without governmental license. To export or re-export this product from a country other than Japan may also be prohibited without a license from that country. Please call an NEC sales representative.

The information in this document is subject to change without notice.
No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Corporation. NEC Corporation assumes no responsibility for any errors which may appear in this document.
NEC Corporation does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from use of a device described herein or any other liability arising from use of such device. No license, either express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Corporation or of others.

Regional Information

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

NEC do Brasil S.A.
Sao Paulo-SP, Brasil
Tel: 011-889-1680
Fax: 011-889-1689

J96. 8

INTRODUCTION

The CC78K0S C Compiler (hereafter referred to as the C compiler) is a C compiler package common to the 78K/0 series of small general-purpose products (referred to as 78K/0S).

The C compiler comes in two models: the normal model, which is specification-compatible with the CC78K0 C Compiler, and the static model, which emphasizes code efficiency. Only the normal model of the C compiler conforms with ANSI-C[Note].

The purpose of the "CC78K0S C Compiler, Language" (hereinafter, referred to as this manual) is to present the basic functions of this C compiler and the language specifications to programmers who will use this C compiler to develop software.

This manual does not describe how to operate this C compiler. Therefore, to use the C compiler after reading this manual, refer to the "CC78K0S Series C Compiler, Operation."

For the architecture of a target device, refer to the relevant User's Manual of the 78K/0 series.

Note ANSI-C is a C language standard consisting of specifications created by the American National Standards Institute (ANSI).

Target devices

This C compiler can be used to develop software for 78K/0 Series small general-purpose microcontrollers.

To use with a target device, the device file (option) for each target type is required.

Target users

This manual is intended for people who have read the user's manual for the microcontroller being developed and are experienced software programmers. No particular knowledge about the C compiler or the C language is needed, but an understanding of software terminology is assumed.

Composition

This manual has the following composition.

CHAPTER 1 OVERVIEW

The general functions, performance, and features of this C compiler are described.

CHAPTER 2 BASIC STRUCTURE OF THE C LANGUAGE

The structure of a C language program and its structural elements are described.

CHAPTER 3 TYPE AND STORAGE CLASS DECLARATIONS

The types, their declarations, and the storage classes used in the C language are described.

CHAPTER 4 TYPE CONVERSION

The automatic type conversion performed by this C compiler is described.

CHAPTER 5 OPERATORS AND EXPRESSIONS

How to describe the operators that can be used in the C language and their precedence are explained.

CHAPTER 6 C LANGUAGE CONTROL STRUCTURES

The control structures of the C language that provide control flow and their uses are described.

CHAPTER 7 STRUCTURES AND UNIONS

An overview of structures and unions and how to use them are described.

CHAPTER 8 EXTERNAL DEFINITIONS

The types of external definitions and how to use them are described.

CHAPTER 9 PREPROCESSING DIRECTIVES

The types of preprocessing directives and how to use them are described.

CHAPTER 10 LIBRARY FUNCTIONS

The library functions and how to use them in the C language are described.

CHAPTER 11 EXTENDED FUNCTIONS

The extended functions for using the target device are described. This function extends the ANSI-C specifications.

CHAPTER 12 REFERENCING ASSEMBLER

How to call assembler programs from a C language program is described.

CHAPTER 13 EFFICIENT COMPILER USE

The procedure for efficiently using this C compiler is described.

APPENDIX

A saddr area label list, compiler output segment names and a function interface list are provided.

How to Read this Manual

The recommended ways to read this manual are described.

- **For beginners to C compilers and the C language**

  Beginners to C compilers and the C language should read the manual in order starting from chapter 1. This manual describes the sequence from the control structures of the C language program to the extended functions. Chapter 1, “Overview,” presents examples of C programs and illustrates references in this manual, so refer to them while reading.

- **For experienced users of C compilers and the C language**

  The language specifications of this C compiler (normal model) conform to ANSI. Therefore, experienced user of C compilers and the C language should start reading from CHAPTER 11 EXTENDED FUNCTIONS. For the static model, be sure to read item (23) Static model in CHAPTER 11 EXTENDED FUNCTIONS. Read CHAPTER 11 EXTENDED FUNCTIONS in conjunction with the user's manual supplied with the target device to be used.

Reference

The *Draft Proposed American National Standard for Information System-Programming Language C* (December 7, 1988).

Convention

The meanings of the symbols used in this manual are described next.

- **In this manual**

...	:	The same format is repeated.
[ ]	:	Characters enclosed by [ ]
“ ”	:	Characters enclosed by “ ”
‘ ’	:	Characters enclosed by ‘ ’
bold character	:	The character itself
_	:	Important places and the input character sequence in the examples are underlined.
⋮	:	Omitted portion of the program [Description]
( )	:	Characters enclosed by ( )
/	:	Delimiter
\	:	Backslash

- **In a structure**

::=	:	The meaning of the lvalue follows.
「」	:	The characters preceding opt can be omitted.
|	:	‘|’ on the left end of the line means ‘or’.

CONTENTS

CONTENTS

List of Figures

List of Tables

CHAPTER 1 OVERVIEW

The CC78K0S C Compiler is a language processing program that converts C language for small general-purpose products of the 78K/0 series (referred as 78K/0S) and source programs described in ANSI-C into machine language. A normal model version, which is compatible with the C compiler CC78K0 for the 78K/0 series excluding small general-purpose products, is provided that offers easy portability. If the static model is used, be sure to read the section listing differences with the normal model. Object files and assembler source files for 78K/0 series small general-purpose products can be obtained from the CC78K0S C compiler.

1.1 The C Language and Assembly Language

Programs and data are needed to run a microcontroller. These are programmed by people and stored in the memory of the microcontroller. Programs and data that can be handled by the microcontroller compose a set of binary numbers called machine language.

The assembly language has a one-to-one correspondence with the mnemonics of the machine language. Because of the one-to-one correspondence with the machine language, assembly language can provide detailed instructions for the computer, such as improving the processing speed during I/O. However, this means that all computer actions must be individually specified. Therefore, the logical structure of the program is difficult to understand at a glance and errors easily arise.

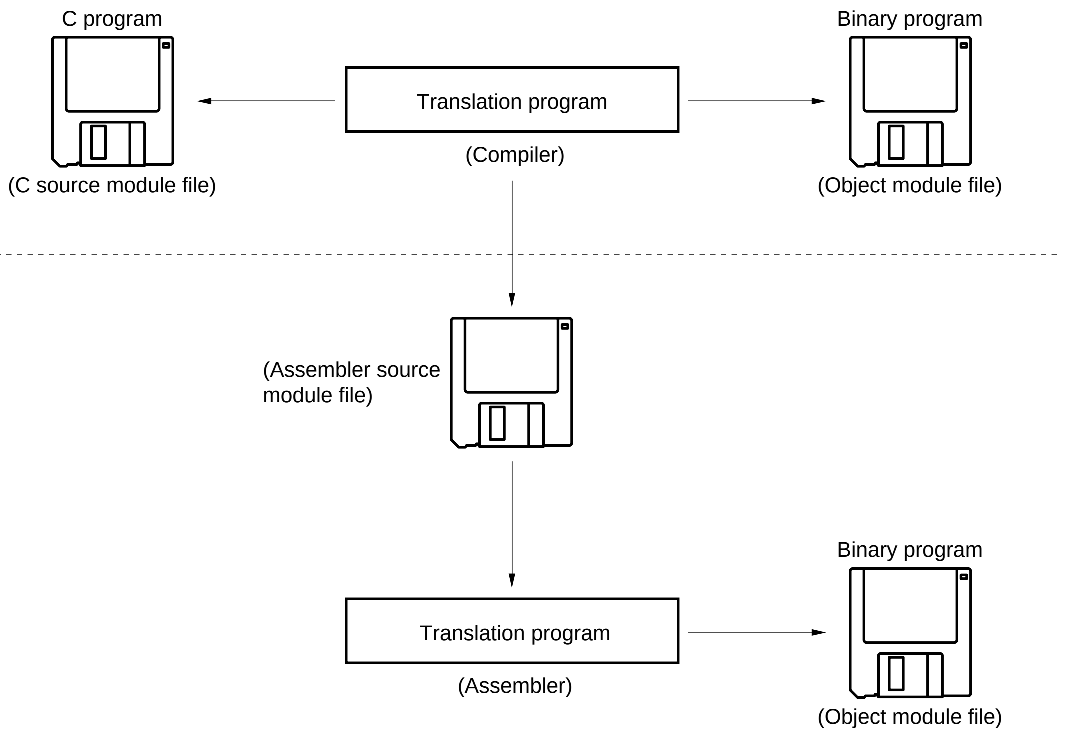
High-level languages were developed to replace assembly languages. One of these is the C language. Consequently, the programmer can write programs without being aware of the computer architecture and more easily understand the logical structure of the program itself than with assembly language.

Since many components (functions) are written to form a program in the C language, the programmer can write a program by combining these components.

A feature of the C language is it’s easy for people to understand. However, the microcontroller cannot understand a program written in the C language. To understand the C language, a program that translates into the machine language compatible with the microcontroller is required. The C compiler is the program that translates C language into machine language.

This C compiler inputs a C source module and outputs an object module and an assembler module. Therefore, the programmer uses the C language to write programs and can revise the program using assembly language to describe detailed instructions about program execution. Figure 1-1, “Compiler Flow,” shows the translation flow in this C compiler.

Figure 1-1. Compilation Flow



Explanation: The C compiler compiles C source module files and generates object module files and assembler source module files.

1.2 Development Procedure Using the C Compiler

Product development using the C compiler requires a linker to link the object module files created by the C compiler, a librarian to create library files, and a debugger to remove bugs from the programs.

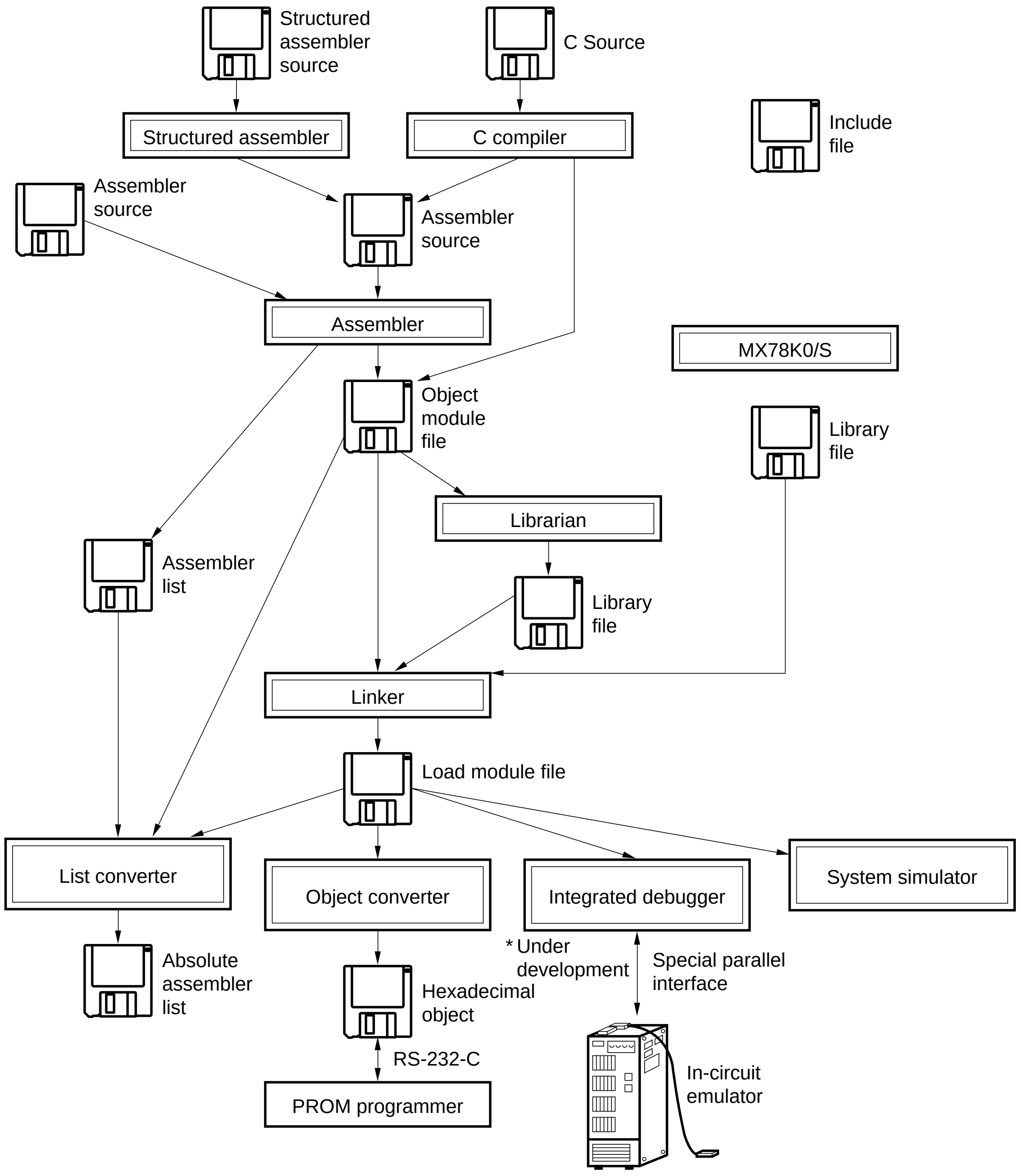
This C compiler requires this software:

- Editor : Creates source module files
- RA78K0S Assembler Package
  - Assembler : Assembles assembler source module files
  - Linker : Links object module files and determines the addresses for relocatable segments
  - Object converter : Converts to hexadecimal files
  - Librarian : Creates library files
- Source debugger : Debugs C source module files

The product development procedure using a C compiler is:

(1) Divide the product into functions.
(2) Write a C source module for each function.
(3) Compile each module.
(4) Build a library of frequently used modules.
(5) Link the modules.
(6) Debug the modules.
(7) Use the object converter to convert to a hexadecimal file.

Figure 1-2. Program Development Procedure Using this C Compiler

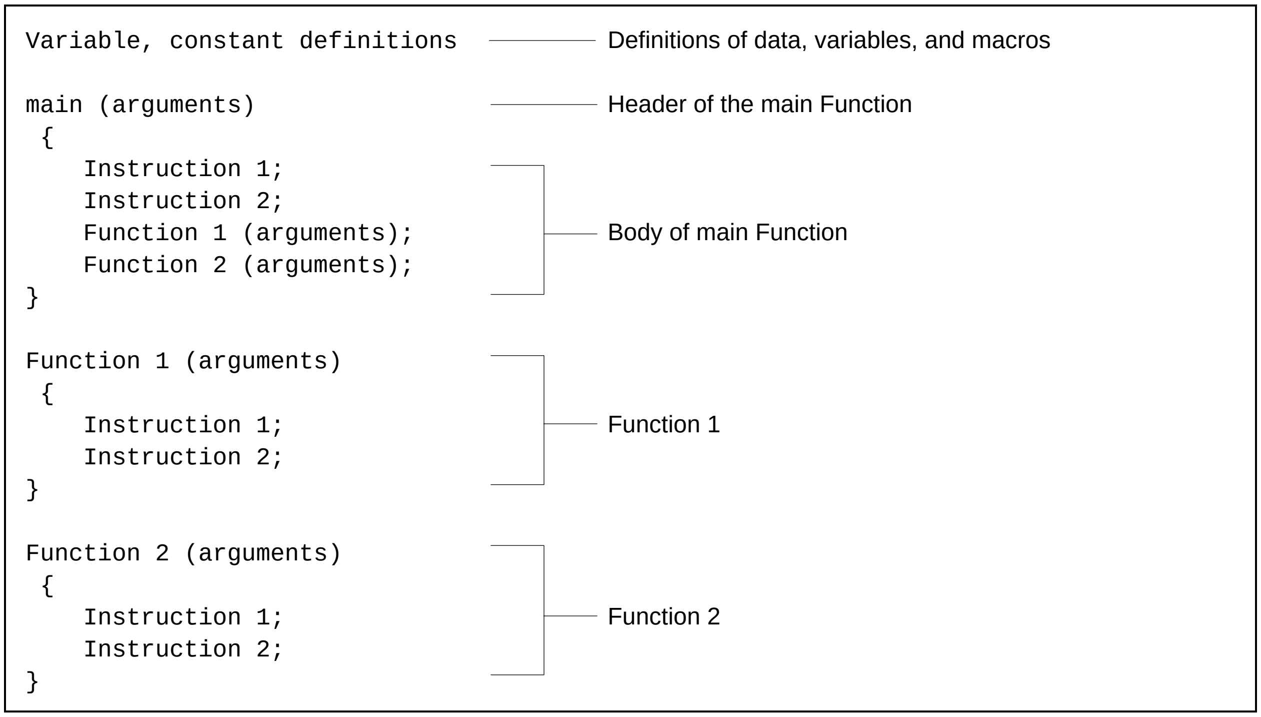


1.3 Basic Structure of a C Program

1.3.1 Program structure

A C program is a collection of functions. Each function is created to have an independent function. Then the functions are collected into one program by the ‘main’ function. The main routine in the C language becomes the ‘main’ function.

A function consists of the header that defines the function names and arguments and the body that specifies the program itself. Next, the structure of a C program is illustrated.



```
Variable, constant definitions

main (arguments)
 {
    Instruction 1;
    Instruction 2;
    Function 1 (arguments);
    Function 2 (arguments);
}

Function 1 (arguments)
 {
    Instruction 1;
    Instruction 2;
}

Function 2 (arguments)
 {
    Instruction 1;
    Instruction 2;
}
```

Definitions of data, variables, and macros
Header of the main Function
Body of main Function
Function 1
Function 2

An actual C source program is shown below.

```
#define TRUE   1    ─┐
#define FALSE  0     ├─#define xxxxxx - Preprocessor directive (macro definition) (6)
#define SIZE   200  ─┘

char mark[SIZE+1]; ──┬─char xxx - Type declaration (1)
main()               └─xx[xx] - Operator (2)
{
    int i, prime, k, count;       ──── int xxx - Type declaration (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); - External definition (5)

                   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);   ────External definition (5)
}

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

    j = i;
    ss = s;
}

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

(1) Type and storage class declarations
The types and the storage classes of the identifiers that indicate objects are declared. For details on the types and storage classes, see Chapter 3, “Type and Storage Class Declarations.”

(2) Operators and expressions
Arithmetic operations, logical operations, and assignments are performed. For details about operators and expressions, see Chapter 5, “Operators and Expressions.”

(3) Control structures
The program flow is specified. The control structure in the C language uses instructions for selection, iteration, and branching. For details about control structures, see Chapter 6, “C Language Control Structures.”

(4) Structures and unions
Structures or unions are declared. A structure is an object containing different types in a contiguous space. A union is an object having a space where different types overlap. For details about structures and unions, see Chapter 7, “Structures and Unions.”

(5) External definitions
A function or external object is defined. A function is one element when a C program is divided into separate functions. A C program is constructed from a collection of functions. For details about external definitions, see Chapter 8, “External Definitions.”

(6) Preprocessing directives
These are instructions for the compiler. ‘#define’ is a directive for the C compiler that replaces the first operand when it appears in the program by the second operand. For details about preprocessing directives, see Chapter 9, “Preprocessing Directives.”

1.4 Before Starting Program Development

Before starting program development, note the following items.

Table 1-1. Optimum Performance of this C Compiler

No.	Item	Limit
1	Compound statement, iteration control statement, selection control statement nesting	45
2	Conditional compiling nesting	255
3	Qualified declarator nesting	12
4	Parentheses nesting in an expression	32
5	No. of significant characters in a macro name	31
6	No. of significant characters in an internal or external symbol name	30 (Note 1)
7	No. of symbols in one source module file	1,024 (Note 2)
8	No. of symbols having block scope in one block	255 (Note 2)
9	No. of macros in one source module file	5,000 (Note 3)
10	Parameters in one function definition and function call	39
11	Parameters in one macro definition and macro call	31
12	No. of characters in one logical source line	509
13	No. of characters in a string literal after linking	509
14	One object size (pointing to data)	65535 bytes
15	#include nesting	8
16	No. of case labels in a switch statement	257
17	No. of source lines in one translation unit	About 3,000
18	No. of source lines that can be compiled without creating a temporary file	About 300
19	Function call nesting	40
20	No. of labels in one Function	33
21	Total size of code, data, stack segments in one object module	65535 bytes
22	Number of members in one structure or union	127
23	Number of enumeration constants in one enumeration	255
24	Nesting of structures or unions in one structure or union	15
25	Nesting of initializer elements	15
26	Number of function definitions in one source module file	400
27	Number of levels of nests of declarators enclosed in parentheses per full declarator.	1

Notes 1. The length of a symbol name can be changed to seven characters by a compiler option.

2. This indicates the maximum that can be processed only in the memory space without using temporary files. When processing cannot take place in the memory space, temporary files are used. This maximum is changed by the file size.

3. This includes macro definitions reserved by the compiler. However, the UNIX version has a maximum of 10,000.

1.5 C Compiler Features

This C compiler provides extended functions that generate non-ANSI CPU code. The extended functions of the C compiler include ones for describing special function registers in the 78K/0S Series in the C language and ones for decreasing the object code size and improving the execution speed. For details about extended functions, see Chapter 11, “Extended Functions.”

The ways to decrease the object code size and improve the execution speed are as follows.

• Calling a function by using the callt area	- callt/_ _callt Function
• Allocating variables to registers	- Register variables
• Allocating variables to the saddr area	- sreg/_ _sreg variables
• Using sfr names	- sfr area
• Creating functions that have no pre- or post-processing (stack frame)	- noauto function, norec/_ _leaf Function
• Specifying assembly language in the C source programs	- ASM statement
• Bit accessing in the saddr and sfr areas	- bit type variables, boolean/_ _boolean type variables
• Storing functions in the callf area	- callf/_ _callf Function
• Enabling the specification of a bit field as an unsigned char	- Bit field declaration
• Direct in-line expansion and output of multiplication code	- Multiply Function
• Direct in-line expansion and output of division code	- Divide Function
• Direct in-line expansion and output of rotate code	- Rotate Function

(1) callt/_ _callt functions

The address of the function to be called is placed in the callt area and the function is called. Compared to ordinary calling, the switching of functions becomes fast. Also the size of the object code can be reduced.

(2) Register variables

These variables can be placed in the registers or the saddr area. Compared to using ordinary variables, the execution speed improves. Also the size of the object code can be reduced.

(3) sreg/_ _sreg variables

These variables can be used in the saddr area. Compared to using ordinary variables, the execution speed improves. Also the size of the object code can be reduced. Even if optional, the variables can be used in the saddr area.

(4) sfr area

The special function register (sfr) can be used in a C source file by using the sfr mnemonic (sfr name).

(5) noauto Function

A function having no pre- or post-processing (stack frame) is generated. When the noauto function is called, the arguments are passed via the registers. Therefore, the execution speed improves and the object code size can be reduced. This function is limited to arguments and automatic variables. For details, see “(5) noauto function” in Chapter 11**.**

(6) norec/_ _leaf functions

A function having no pre- or post-processing (stack frame) is generated. When the norec or _ _leaf function is called, variables are passed via the registers. The automatic variables used in the norec or _ _leaf function are allocated to registers or the saddr area. Therefore, improved execution speed and reduced the object code size are possible. These functions are limited to arguments and automatic variables. Also function calls cannot be made from this function. For details, see “(6) norec function” in Chapter 11.

(7) bit, boolean, and _ _boolean type variables

Variables having a one-bit storage area are created. By using bit, boolean, and _ _boolean type variables, the saddr area can be accessed in bits.
The boolean and _ _boolean type variables have the same function and use as a bit type variable.

(8) ASM statement

Assembler source described by the user is embedded in the assembler source file output by C compiler.

(9) Kanji

Kanji can be written in comments in the C source file. Shift JIS code, EUC code, or no kanji code can be selected for the kanji code.

(10) Interrupt functions

A vector table is generated and the object code for interrupts is output. This allows interrupt functions to be described in a C source level.

(11) Interrupt function qualifiers

By using qualifiers, vector table settings and interrupt function definitions can be described in separate files.

(12) Interrupt operation

Interrupt disable instructions and interrupt enable instructions are embedded in objects.

(13) CPU control instructions

The following instructions are embedded in objects.

halt
stop
nop

(14) Absolute address access functions

Code that accesses the normal memory space in an object does not perform a function call, but is directly expanded in line and output to generate the object file.

(15) Bit-field declarations

Defining the type of a bit field as unsigned char results in memory savings, object code reduction, and faster execution speed.

(16) Change compiler output section name Function

By changing the compiler output section name, the section can be independently located by the linker.

(17) Binary number description Function

Binary numbers can be described in the C source.

(18) Change module name Function

The name of the object module can be freely changed in the C source.

(19) Rotate functions

Code that rotates the value of an expression in an object is directly expanded in line and output.

(20) Multiplication Function

Code that multiplies the value of an expression in an object is directly expanded in line and output. This function reduces the object code size and improves the execution speed.

(21) Division Function

Code that divides the value of an expression in an object is directly expanded in output. This function reduces the object code size and improves the execution speed.

(22) Data insertion Function

Constant data is inserted at the current address. Without using assembler descriptions, special data or instructions can be embedded in the code space.

[MEMO]

CHAPTER 2 BASIC STRUCTURE OF THE C LANGUAGE

This chapter describes the structural elements of a C source module file. A C source module file consists of ‘tokens.’ The tokens are:

Keywords	Identifiers	Constants
String literals	Operators	Punctuators
Header names	Preprocessing numbers	Comments

The tokens used in the following example description of a C program are indicated.

```
#include ”expand.h”

extern bit data1;
extern bit data2;                extern - Keyword
                                 data1, data2 - Identifiers
void main()                      void - Keyword
{
       data1 = 1;                1 - Constant
       data2 = 0;                0 - Constant

       while (data1){            while - Keyword
                 data1 = data2;  {} - Punctuators
                 testb();        = - Operator
       }
       if (data1 && data2){      if - Keyword
                                 && - Operator
       }         chgb();         () - Operator
}
lprintf(s, i)                    lprintf - Identifier
char *s;                         char, int - Keyword
int i;                           s, i - Identifier
{
       int   j;
       char *ss;                 * - Operator
       j=i;
       ss=s
}
                           :
```

(1) Character set

The character sets used in a C program are the set of source characters that describe the source files and the set of execution characters that are interpreted by the execution environment.

The value of a character in the execution character set is 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 (Note 1)
        ! ” # % & ’ ( ) * + , - . / :
        ; < = > ? [ ¥ ] ^ _ { | } ~
```

Control characters indicate the space, vertical tab, horizontal tab, and form feed.

Note 1. In the PC DOS, there is no yen symbol ‘¥,’ but there is a backslash ‘\’.

(2) Multibyte characters

The source character set can use multibyte characters in the extended character set (i.e., comments). The execution character set can use shift JIS kanji code or EUC kanji code multibyte characters.

(3) Escape sequences

Non-graphic characters like warnings and form feeds are represented by escape sequences. An escape sequence consists of the yen symbol ¥ and one letter.

The escape sequences that represent non-graphic characters are shown next.

Table 2-1. Escape Sequences

Escape Sequence	Represents	Character Code
¥a	Warning	07H
¥b	Backspace	08H
¥f	Formfeed	0CH
¥n	New line	0AH
¥r	Carriage return	0DH
¥t	Horizontal tab	09H
¥v	Vertical tab	0BH

2.1 Keywords

Since the following tokens are used as keywords by the compiler, they 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
```

To implement extended functions in this C compiler, the following tokens have been added as keywords. (If these tokens contain uppercase letters, they are not considered to be keywords.)

Specifying the -ZA option disables keywords that do not begin with an underscore (_ _).

`callt/_ _callt`	- callt function declaration
`sreg/_ _sreg`	- sreg variable declaration
`noauto`	- noauto function declaration
`norec/_ _leaf`	- norec function declaration
`bit`	- bit type variable declaration
`boolean/_ _boolean`	- boolean type variable declaration
`_ _interrupt`	- hardware interrupt Function
`_ _asm`	- asm statement

2.2 Identifiers

The identifiers are:

Functions
Objects
Structure, union, and enumeration tags
Structure, union, and enumeration members
typedef names
Label names
Macro names

An identifier is represented by lowercase letters, uppercase letters, digits, and the underscore.

```
_ (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
```

The first character in an identifier cannot be a digit. Also an identifier must not be a keyword.

2.2.1 Identifier scope

The valid range where an identifier can be used is determined by the position where it was declared. The valid range of an identifier is called the scope of the identifier.

Identifiers have these scopes:

Function scope
File scope
Block scope
Function prototype scope

```
extern boolean data1, data2;     ———— data1, data2 - File scope

void main()
{
       int cot;                  ———— cot - Function scope
       data1 = 1;
       data2 = 0;

       while(data1){
                 data1 = data2;
                 j1:             ———— j1 - Block scope
                 testb(cot);
       }
}

void testb(int x)                ———— x - Function prototype scope
{

      .
      .
      .
```

(1) Function scope

Function scope indicates the entire function. An identifier having function scope can be referenced from anywhere in the function.

An identifier having function scope is only a label name.

(2) File scope

File scope indicates the entire translation unit.

An identifier declared outside a block or parameter list has file scope. An identifier having file scope can be referenced from anywhere in the program.

(3) Block scope

The block scope indicates the range until a specific block (enclosed by braces { }) ends.

An identifier declared in a block or a parameter list has block scope. An identifier with block scope is valid in the specified block.

(4) Function prototype scope

This indicates the range until the end of the declared function.

An identifier declared in the parameter list in a function prototype has function prototype scope. An identifier having function prototype scope is valid in the specified function.

2.2.2 Identifier linkage

Identifier linkage is declaring more than one identifier with different or the same scopes so they can be referenced as the same object or function. These identifiers are considered to be identical by linkage.

The kinds of identifier linkage are external linkage, internal linkage, and no linkage.

(1) External linkage

External linkage links the set of translation units and libraries that form the entire program.

External linkage occurs in the following cases.

- Declared function without a storage-class specifier
- No storage-class specifier for the identifier to be referenced in an object or function declared as extern
- Objects that have file scope and no storage-class specifier

(2) Internal linkage

Internal linkage links inside one translation unit.

Internal linkage occurs in the following case.

- Object or function that has file scope and includes the static storage-class identifier

(3) No linkage

An identifier with no linkage is a unique entity.

Examples of no linkage are:

- Identifier for something other than an object or Function
- Identifier that declares a function parameter
- Identifier of an object that does not have an extern storage-class specifier in the block

2.2.3 Name spaces of identifiers

All identifiers are classified into the following ‘name spaces.’

• Label name	- Disambiguated by the label declaration
• Tags for unions, structures, and enumerations	- Disambiguated by the struct, union, and enum keywords
• Members of structures and unions	- Disambiguated in an expression by the . and -> operators
• Ordinary identifiers (identifiers other than the above)	- Declared as ordinary declarators or enumeration constants

2.2.4 Storage durations of objects

Each object has a ‘storage duration’ that determines its lifetime. The two storage durations are static and automatic.

(1) Static storage duration

An object having static storage duration has its storage reserved before execution. The reserved storage is initialized only once. A static object exists while the program is executing and retains its last stored value.
Objects having static storage duration are:

- Objects having external linkage
- Objects having internal linkage
- Objects declared by the storage-class specifier static

(2) Automatic storage duration

An object having automatic storage duration has its storage reserved upon entering the block where it is declared. If entering at the beginning of the block and initialization is specified, the object is initialized. If the block is entered by jumping to a label, there is no initialization.
The storage for an object having automatic storage duration is not reserved when the execution of the declared block ends.
Objects having automatic storage duration are:

- Objects with no linkage
- Objects not declared with the storage-class specifier static

2.2.5 Types

The type determines the meaning of the value stored in the object. The type is specified in the identifier that declares the object.

A type is classified as one of the following three types based on the declarator.

- Object types - Types that describe objects
- Function types - Types that describe functions
- Incomplete types - Types that describe objects that do not have size information

The types are:

- Basic types (arithmetic types)
  - Integer types
    - char type
    - Signed integer types
      - signed char
      - short int
      - int
      - long int
    - Unsigned integer types
    - Enumerated types
  - Floating types
    - float, double, log double

      * Normal model only supported

- Character types
  - char
  - signed char
  - unsigned char

- Incomplete types
  - Array, structure, union, and void types that do not specify the object size

- Derived types
  - Aggregate types
    - Array types
    - Structure types
  - Union types
  - Derived declarator types
    - Array types
    - Function types
    - Pointer types

- Scalar types
  - Basic types
    - char type
    - Signed integer types
    - Unsigned integer types
  - Pointer types

(1) Basic types

The basic types, also called arithmetic types, consist of integer types. The integer types are classified into char type, signed integer type, unsigned integer type, and enumeration type.

(a) Integer types

There are four integer types. The value of an integer type is represented by the binary numbers 0 and 1.

- char type
- Signed integer type
- Unsigned integer type
- Enumeration type

(i) char type

The char type is large enough to store any character in the execution character set.

The value of a character stored in a char object is positive. Other characters are handled as signed integers. If an overflow occurs when storing, the overflow is ignored.

(ii) Signed integer types

The four signed integer types are:

- signed char
- short int
- int
- long int

An object declared with the signed char type has the same amount of storage as a char without a qualifier.

An int object that has no qualifier has the natural size suggested by the CPU architecture of the execution environment.

The unsigned integer types that correspond to the signed integer types use the same amount of storage. Positive numbers having signed integer type are a subset of unsigned integer type.

(iii) Unsigned integer type

An unsigned integer type is represented by the unsigned keyword.

Computations involving unsigned integer types do not overflow. If values that cannot be represented by integer types occur in computations involving unsigned integer types, the computation result is reduced modulo the number that is the sum of the largest number that can be represented by the unsigned integer type plus one.

(iv) Enumeration type

An enumeration is a set of named integer constants. It is formed by a list.

Table 2-2. Basic Types

Type	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

- signed can be omitted. However, when signed is omitted for char type, signed char or unsigned char is determined based on the conditions (options) during compilation.
- short int and int are handled as different types with the same range of values.
- unsigned short int and unsigned int are handled as different types with the same range of values.

(2) Character types

There are three character types.

- char
- signed char
- unsigned char

(3) Incomplete types

There are four incomplete types.

- Array of objects of an unknown size
- Structures
- Unions
- void types

(4) Derived types

A derived type can be derived from basic, enumeration, and incomplete types.

- Aggregate type
- Union type
- Derived declarator type

(a) Aggregate type

The two aggregate types are the array types and the structure types. An aggregate type is a collection of concatenated member objects. The array type is also included in the derived declarator type.

(i) Array type

The array type is a set of member objects called the element type. Since the array type is derived from one member object, it is called an array type derivation. Therefore, all of the member objects have the same size storage. If there is an array type derivation based on the T element type, the resulting type is said to be an "array of T."

(ii) Structure type

The structure type is a set of member objects having different sizes. Each member object can be specified by its name.

(b) Union type

The union type is a set of overlapping member objects. Each member object has a different size and name, and can be individually specified.

(c) Derived declarator type

There are three derived declarator types.

- Array type
- Function type
- Pointer type

(i) Array type

The derived declaration of the array type is called an array type derivation.

(ii) Function type

The function type represents a function with the specified return type. The function type is characterized by the type of its return value, number of parameters, and types of the parameters. A function is derived from its return type. If the return type is T, that function is said to be a "function returning T." Creating a function based on the return type is called function derivation.

(iii) Pointer type

The pointer type is constructed from a function type, an object type, or an incomplete type called the referenced type. The pointer type describes an object. The value representing the object is used to reference the entity having the referenced type.
The pointer type constructed from referenced type T is called the "pointer to T." Constructing a pointer type from the referenced type is called pointer derivation.

(5) Scalar types

Arithmetic types and pointer types, which are basic types, are generally referred to as scalar types. The scalar types are:

- char type
- Signed integer type
- Unsigned integer type
- Pointer type

2.2.6 Compatible types and composite types

(1) Compatible types

If two types are the same, they are said to be compatible types. For example, two structures, unions, or enumerations declared in separate translation units are compatible types if they have the same number of members, member names, and compatible member types. Two structures or unions must have members arranged in the same order. Two enumerations must have members with the same values.
All declarations related to the same object or function must have a compatible type.

(2) Composite types

A composite type is a type that is compatible with two types. A composite type is constructed from two compatible types. The following situations arise for the composite type.

- If one type is an array with a known size, the composite type is an array having the same size.
- If one type is a function type with a parameter type list (function prototype), the composite type is a function prototype with a parameter type list.
- If both types are parameter type lists, the parameter type of the composite parameter type list is the composite type of the corresponding parameters.

These rules apply recursively to the two types that are derived.

[Composite Type Example]

Two declarations having file scope are:

```
      int f(int(*)(), double(*)[3]);
      int f(int(*)(char*), double(*)[]);
```

The composite type of the functions is:

```
      int f(int(*)(char *), double(*)[3]);
```

2.3 Constants

A constant is a preset value. The type of each constant is determined by the specified format and value. There are four types of constants.

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

2.3.1 Floating constants

Floating constants have this syntax:

```
floating-constant ::=
          fractional-constant exponent-part floating-suffix
        | digit-sequence exponent-part floating-suffix
        fractional-constant ::=
            digit-sequence . digit-sequence
            digit-sequence .
        exponent-part ::=
              e sign digit-sequence
            | E sign digit-sequence
            sign ::=
                +
                | –
        digit-sequence ::=
              digit
            | digit-sequence digit
        floating-suffix ::=
              f
            | l
            | F
            | L
```

2.3.2 Integer constants

Integer constants are integer values specified beforehand. There are three types of integer constants.

- Decimal constants
- Octal constants
- Hexadecimal constants

Integer constants have this syntax:

```
integer-constant ::=
          decimal-constant ⌈integer-suffix⌋
        | octal-constant ⌈integer-suffix⌋
        | hexadecimal-constant ⌈integer-suffix⌋
        decimal-constant ::=
              nonzero-digit
            | decimal-constant digit
            nonzero-digit ::= one of
                1 2 3 4 5 6 7 8 9
        octal-digit ::=
              0
            | octal-digit octal-digit
            octal-digit ::= one of
                0 1 2 3 4 5 6 7
        hexadecimal-constant ::=
              0x hexadecimal-digit
            | 0X hexadecimal-digit
            | hexadecimal-constant hexadecimal-digit
            hexadecimal-digit ::= one of
                0 1 2 3 4 5 6 7 8 9
                a b c d e f
                A B C D E F
        integer-suffix ::=
              unsigned-suffix ⌈long-suffix⌋
            | long-suffix ⌈unsigned-suffix⌋
            unsigned-suffix ::= one of
                u U
            long-suffix ::= one of
                l L
```

The type of an integer constant is considered to be the first type under the ‘Type That Can Be Represented’ in the following list.

However, the type of an unsuffixed constant can be changed to char or unsigned char by the compilation conditions (options).

(Integer Constant)	(Type that can be Represented)
• Unsuffixed decimal	int, long int, unsigned long int
• Unsuffixed octal, hexadecimal	int, unsigned int, long int, unsigned long int
• Suffixed by u or U	unsigned int, unsigned long int
• Suffixed by l or L	long int, unsigned long int
• Suffixed by u or U and l or L	unsigned long int

(1) Decimal constants

A decimal constant is an integer with base 10. To specify, begin with a digit other than 0 followed by the digits 0 to 9.

(2) Octal constants

An octal constant is an integer with base 8. To specify, begin with a 0 followed by the digits 0 to 7.

(3) Hexadecimal constants

A hexadecimal constant is an integer with base 16. To specify, begin with 0x or 0X followed by the decimal digits and the letters a (or A) to f (or F), which represent the numbers 10 to 15.

2.3.3 Enumeration constants

An enumeration constant is an element in an enumeration type variable. An enumeration type variable can only have a specific value represented by an identifier. An enumeration constant is used to represent the value of an enumeration type variable.

The identifier that declared an enumeration constant becomes the int type. The enumeration constant is represented by an identifier.

2.3.4 Character constants

A character string constant is a character string of one or more characters enclosed by single quotes, such as ‘X’ or ‘ab’.

Character constants have this syntax:

```
character-constant ::=
            ‘c-char-sequence’
      c-char-sequence ::=
             c-char
            | c-char-sequence c-char
            c-char ::=
                 any character in source character set of the machine except the single quote ‘,
                 backslash \, or new-line character
                 | escape sequence
      escape-sequence ::=
             simple-escape-sequence
            | octal-escape-sequence
            | hexadecimal-escape-sequence
            simple-escape-sequence ::= one of
                 \’ \” \? \\
                 \a \b \f \n \r \t \v
```

```
octal-escape-sequence ::=
      \ octal-digit
     | \ octal-digit octal-digit
     | \ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence ::=
      \x hexadecimal-digit
     | hexadecimal-escape-sequence hexadecimal-digit
```

2.4 Strings

A string literal is a sequence of zero or more characters enclosed by double quotes, as in “xxx”.

A single quote (‘) is represented by itself or the escape sequence \’. A double quote (“) is represented by the escape sequence \”.

An array element has type char.

String literals have this syntax:

```
string-literal ::=
            “⌈s-char-sequence⌋“
      s-char-sequence ::=
              s-char
            | s-char-sequence s-char
            s-char ::=
                 any character in the source character set of the machine except the double quote “,
                 backslash \, and \n
                 | escape-sequence
```

2.5 Operators

An operator specifies how to perform an evaluation. The entity acted on by an operator is called the operand. An operand is evaluated in the manner specified by the operator. Values, designators, side effects, or their combinations are generated.

Operators have this syntax:

operator ::= one of

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

The operators [ ], ( ), and ?: must occur in pairs. They may contain expressions.

# and ## are only used in the macro definitions of preprocessing directives.

2.6 Punctuators

A punctuator is a symbol that has independent syntactic and semantic significance. However, values cannot be generated. The punctuators are:

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

The punctuators [ ], ( ), and { } may contain expressions, declarations, or statements. However, these punctuators must always be used in pairs.

The # punctuator is only used in preprocessing directives.

2.7 Header Names

The header name indicates the external source file name. This is only used in the #include preprocessing directive.

Header names have this syntax:

```
header-name ::=
        <h-char-sequence>
      | “q-char-sequence”
      h-char-sequence ::=
           h-char
         | h-char-sequence h-char
         h-char ::=
              any character in the source character set of the machine except the new-line
              character and >
      q-char-sequence ::=
           q-char
         | q-char-sequence q-char
         q-char ::=
              any character in the source character set of the machine except the new-line character
              and “
```

2.8 Preprocessing Numbers

A preprocessing number is the number before conversion to an integer constant. In the preprocessing number stage, the type and value are not maintained.

Preprocessing numbers have this syntax:

```
pp-number ::=
          digit
        | . digit
        | pp-number digit
        | pp-number nondigit
        | pp-number e sign
        | pp-number E sign
        | pp-number .
```

A preprocessing number begins with a digit or a digit preceded by a period (.).

Subsequent digits are any of the following.

Letters
Underscore
Digits
Period
e+
e–
E+
E–

2.9 Comments

A comment is a note inserted in the C source module. A comment statement begins with /* and ends with */. This C compiler can identify multibyte strings and kanji can be used.

CHAPTER 3 TYPE AND STORAGE CLASS DECLARATIONS

This chapter describes the data types, the function types, declarations that are used in the C language, and their scope. A declaration specifies the interpretation of identifiers or identifier sets and their attributes. A declaration that also reserves storage space for the object or function named by the identifier is a definition.

Declarations have this syntax:

```
declaration ::=
                  declaration-specifiers⌈init-declarator-list⌋ ;
          declaration-specifiers ::=
                    storage-class-specifier⌈declaration-specifiers⌋
                  | type-specifier⌈declaration-specifiers⌋
                  | type-qualifier⌈declaration-specifiers⌋
          init-declarator-list ::=
                    init-declarator
                  | init-declarator-list , init-declarator
                  init-declarator ::=
                            declarator
                          | declarator = initializer
```

Here is an example declaration.

```
#define  TRUE   1
#define  FALSE  0
#define  SIZE   200
void main()
{
   auto int l, prime, k;

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

A declaration specifier is a sequence of specifiers that indicate a portion of the entity types indicating the storage duration and declarators. An init declarator list is a sequence of declarators separated by commas. Each declarator can hold additional type information or initializers, or both.

If the identifier for some object is declared with no linkage, the type for that object must be terminated by that declarator, or the init declarator if there is an initializer.

The type for an object with no linkage must be terminated by a declarator or init declarator.

3.1 Storage-Class Specifiers

A storage-class specifier indicates the storage class of an object. The storage class indicates the storage location of the value holding the object and the object scope. The declaration specifier in one declaration can describe only one storage class specifier. The five storage class specifiers are:

- typedef
- extern
- static
- auto
- registe

(1) typedef

The typedef specifier becomes a storage class specifier only in a suitable syntax. The typedef specifier declares an synonym for the specified type.
For details about the typedef specifier, see Section 3.6, “typedef**.**”

(2) extern

The extern specifier indicates an external variable.

(3) static

The static specifier indicates that the object has static storage duration.
An object having static storage duration reserves storage before program execution. The value to be stored is initialized only once. The object exists during program execution and the last value stored is saved.

(4) auto

The auto specifier indicates that the object has automatic storage duration.
An object having automatic storage duration reserves storage during execution. When entering at the beginning of a block and initialization is specified, the object is initialized. When entering by jumping to a label within a block, there is no initialization.
An object space having automatic storage duration is not maintained after the execution of the declared block ends.

(5) register

The register specifier indicates that the object is allocated to the CPU register. In this C compiler, register specifiers are allocated to the registers and saddr space. For details on register variables, see Chapter 11, “Extended Functions**.**”

3.2 Type Specifiers

Type specifiers indicate the type of an object. The type specifiers are as follows:

- void
- char
- short
- int
- long
- float*
- double*
- long double*
- signed
- unsigned
- struct-or-union-specifier
- enum-specifier
- typedef-name
- bit, boolean, _ _ boolean

* Only normal model can be specified.

Each type specifier is listed below. There are no other type specifications. The bit, boolean, and _ _ boolean types are compiler independent types.

• void	- Null value set
• char	- Size that can store the basic character set
• signed char	- Signed integer (–128 to +127)
• unsigned char	- Unsigned integer (0 to 255)
• short, signed short, short int, signed short int	- Signed integer (–32,768 to +32,767)
• unsigned short, unsigned short int	- Unsigned integer (0 to 65,535)
• int, signed, signed int	- Signed integer (–32,768 to +32,767)
• unsigned, unsigned int	- Unsigned integer (0 to 65,535)
• long, signed long, long int, signed long int	- Signed integer (–2,147,483,648 to +2,147,483,647)
• unsigned long, unsigned long int	- Unsigned integer (0 to 4,294,967,295)
• float	- Floating-point (1.17549435E-38F to 3.40282347E+38F)
• double	- Double precision floating-point (1.17549435E-38F to 3.40282347E+38F)
• long double	- Extended precision floating-point (1.17549435E-38F to 3.40282347E+38F)
• struct-or-union-specifier	- Member object set
• enum-specifier	- Set of int constants
• typedef-name	- Synonym for the specified type
• bit, boolean, _ _ boolean	- Integer that be represented by one bit (0 and 1)

Type specifiers delimited by commas have the same size.

3.2.1 Structure specifiers and union specifiers

Structure specifiers and union specifiers indicate a set of named member objects. Each member object can have different types.

Structure specifiers and union specifiers have this syntax:

```
struct-or-union-specifier ::=
            struct-or-union-specifier identifier {struct-declaration-list}
          | struct-or-union-specifier identifier
          struct-or-union ::=
                  struct
                | union
          struct-declaration-list ::=
                  struct-declaration
                | struct-declaration-list struct-declaration
                struct-declaration ::=
                        specifier-qualifier-list struct-declarator-list;
                      specifier-qualifier-list ::=
                              type-specifier⌈specifier-qualifier-list⌋
                            | type-qualifier⌈specifier-qualifier-list⌋
                      struct-declarator-list ::=
                              struct-declarator
                            | struct-declarator-list, struct-declarator
                            struct-declarator ::=
                                    declarator
                                  | ⌈declarator⌋ : constant-expression
```

Structure declaration example

```
struct tnode

         int count;
         struct tnode *left, *right;

};
```

The structure declaration declares a set of different types as one object to the compiler. Each type is called a member object and can be assigned a name. The member objects are reserved in contiguous space in the declared order.

A union specifier declares a set of different types as one object to the compiler. The member objects in the union are reserved in overlapping space.

- Member object declaration
  The member objects are declared by a structure declaration list. The types of the member objects can be any type other than the incomplete or function type. Also member objects can have bit fields.

- Bit field

  A bit field is an area for an integral type composed of the specified number of bits. The int, unsigned int, and signed int types can be specified in a bit field. (In the CC78K Series, the char, unsigned char, and signed char types can also be specified, but the signed type is not supported so this type is always considered to be the unsigned type.) The most-significant bit in an int bit field and signed bit field that is not a qualifier is considered to be the sign bit. In this compiler, the allocation direction of a bit field can be changed by the compilation conditions (options). (For details, see Chapter 11, “Extended Functions.”)

  For multiple bit fields, if sufficient space remains in the same memory unit, the next bit field is packed in the next bit and entered. When an unnamed bit field with zero width is positioned, the next bit field in the same memory unit is not packed. An unnamed bit field is declared without a declarator and consists of only a colon and the width.

```
struct data {
        unsigned inta:2;
        unsigned intb:3;
        unsigned intc:1;

}no1;
```

3.2.2 Enumeration specifiers

An enumeration specifier indicates ordered objects. An object declared by an enumeration specifier is declared as a constant of type int.

Enumeration specifiers have this syntax:

```
enum-specifier ::=
        enum⌈identifier⌋{enumerator-list}
      | enum identifier
      enumerator-list ::=
            enumerator
          | enumerator-list, enumerator
          enumerator ::=
               enumeration-constant
               enumeration-constant = constant-expression
```

The objects are declared in the enumerator list. The objects are defined by starting with 0 in the declared order. Subsequent objects are defined by adding 1 in order. A constant value can be specified by using an equal sign (=).

In the following example, hue is an enumeration tag, col is an object having this type, and cp is a pointer to an object having this type. In this declaration, the enumeration values become {0, 1, 20, 21}.

```
enum hue
      chartreuse,
      burgundy,

      claret=20,
      winedark
};
  /*…*/
enum hue col, *cp;
col = claret;
cp = &col;
/*…*/(*cp != burgundy)/*…*/
```

3.2.3 Tags

A tag is the name assigned to a structure, union, or enumeration type. The tag has the declared type. An object having the same type can be declared by using the tag.

The identifier in the following declarations is the tag.

struct-or-union identifier {structure-declaration-list}
or
enum identifier {enumerator-list}

A tag holds the contents of the structure, union, or enumeration declared by the list. The tag can be specified once, and the list specifying the structure, union, or enumeration omitted. In subsequent declarations, structures identical to the list having the tag result. Subsequent declarations in the same scope must omit the braced list.

This type specifier

struct-or-union identifier

has undefined contents, so the structure or union has incomplete type. This tag is only used when the size of the object is not required. By defining the tag contents in the same scope, the type becomes complete.

In the following example, tnode specifies a structure containing an integer and two pointers to objects having the same type.

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

The following example declares s to be an object with the type indicated by the tag and sp as a pointer to an object having the type indicated by the tag. Based on this declaration, the expression sp->left is a pointer to the struct tnode of the left object indicated by sp.

s.right ->count indicates count which is a member of the right struct tnode of s.

```
typedef struct tnode TNODE;
struct tnode
          int count;
          TNODE *left, *right;
};
TNODE s[5], *sp;
sp = s;
sp->left = s[0];
sp->right = s[1];
s[0].right->count = 2;
```

The following example specifies structures that have a pointer to the other's structure.

```
struct s1 {struct s2 *s2p; /*…*/}; /* D1 */
struct s2 {struct s1 *s1p; /*…*/}; /* D2 */
```

Within the enclosed scope, if s2 has already been declared as a tag, declaration D1 does not reference the s2 tag that is declared in D2, but an entity declared previously. To eliminate this dependence on context, a meaningless struct s2 declaration is inserted before D1. This declares a new s2 tag within the internal scope. The following D2 declaration completes the new type specification.

3.3 Type Qualifiers

The two type qualifiers are const and volatile. They only affect the lvalues.

An object defined with a const qualifier type by using an lvalue having a non-const qualifier type cannot be modified. An object defined with a volatile qualifier type by using an lvalue having a non-volatile qualifier type cannot be referenced.

An object having volatile-qualified type can be changed in ways unknown to the compiler or can have other unknown side effects. Therefore, an expression referencing this object must be strictly evaluated according to the ordering rules on how to execute a program written in the C language. The values stored last in the object at all sequence points must agree with the ones selected by the program, except for ones changed by unknown factors.

When a type qualifier appears in an array type specification, the type qualifier qualifies the elements in the array and not the array. A type qualifier cannot be included in a function type specification. However, callt, callf, noauto, and norec described in section 2.1, “Keywords,” can be included as type qualifiers.

Two compatible qualifier types must be qualifier types with the same compatible type. The order of the type qualifiers in a specifier list or qualifier list is not affected by the specified type. Another type qualifier is sreg (qualifier independent of this compiler) described in section 2.1, “Keywords.”

The next example has a real_time_clock that can be modified by the hardware, but operations like assignment, increment, and decrement cannot be performed.

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

This example shows the case where the type qualifier qualifies the aggregate type.

```
const struct s {int mem;} cs = {1};
struct s ncs;            /* The ncs object can be modified. */
typedef int A[2][3];
const A a = {{4,5,6},{7,8,9}};   /* Array of const int arrays */
int *pi;
const int *pci;

ncs = cs;                /* Valid */
cs = ncs;                /* Violates constraints on lvalues that can be modified for = */
pi = &ncs.mem;           /* Valid */
pi = &cs.mem;            /* Violates type constraints for = */
pci = &cs.mem;           /* Valid */
pi = a[0];               /* Invalid: a[0] has the type ‘const int*’ */
```

3.4 Declarators

A declarator declares one identifier. In particular, pointer declarators, array declarators, and function declarators are described here. Functions or objects with identifier scope, storage duration, and type are determined by the declarators.

Declarators have this syntax:

```
declarator ::=
          ⌈pointer⌋ direct-declarator
     pointer ::=
            *⌈type-qualifier-list⌋
          | *⌈type-qualifier-list⌋ pointer
          type-qualifier-list ::=
                 type-qualifier
               | type-qualifier-list type-qualifier
     direct-declarator ::=
            identifier
          | (declarator)
          | direct-declarator [⌈constant-expression⌋]
          | direct-declarator⌈parameter-type-list⌋
          | direct-declarator [⌈identifier-list⌋]
          parameter-type-list ::=
                 parameter-list
               | parameter-list,...
               parameter-list ::=
                      parameter-declaration
                    | parameter-list, parameter-declaration
                    parameter-declaration ::=
                           declaration-specifiers declarator
                         | declaration-specifiers⌈abstract-declarator⌋
          identifier-list ::=
                 identifier
               | identifier-list, identifier
```

3.4.1 Pointer declarators

A pointer declarator indicates that the identifier to be declared is a pointer. A pointer points to a place where a value is stored.

By declaring the type of the object to be pointed at, the offset is obtained by computing the pointer variable.

Pointer declaration

T D1

T: Declaration identifier specifying the T1 type (i.e., int)
D1: Declarator including the D identifier (i.e., ident)

Here, D1 has this format.

*「type-qualifier-list」 D

This declaration makes D a pointer qualified by the type qualifier.

The next two declarations show a “variable pointer to a constant value” and a “constant pointer to a variable value.”

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

The first declaration means that the contents of the const int pointed to by ptr_to_constant is not changed, but the ptr_to_constant itself may be changed to point to another const int. Similarly, in the second declaration, the contents of the int pointed to by constant_ptr can be changed, but constant_ptr itself always points to the same position.

The declaration of the constant point constant_ptr can be clarified by including the definition for the type “pointer to int.”

The next example declares constant_ptr as an object with type “const-qualified pointer to int.”

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

3.4.2 Array declarators

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

Array declaration

T D1

T: Declaration specifier specifying the T1 type (i.e., int)
D1: Declarator including the identifier D (i.e., ident)

D1 has this format.

D [⌈constant-expression⌋]

This declaration makes D1 an array having the size of type T1. The value of a constant expression becomes the number of elements in the array. The constant expression is an integer constant expression holding values greater than 0. When a constant expression is not specified in the array declaration, the array is the incomplete type.

The next example declares fa[], a char type array of 11 elements, and afp[], an array of pointers to char type of 17 elements.

```
char fa[11], *afp[17];
```

The next example declares that the first declaration of x is a pointer to type int. The second declaration declares that y is an array of type int with unknown size that is declared at another location.

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

3.4.3 Function declarators (including prototype declarations)

A function declarator declares the value returned by a function and the argument types.

Function declaration

T D1

T :	Declaration specifier specifying the T1 type (i.e., int)
D1 :	Declarator including the identifier D (i.e., ident)

D1 has this format.

D (parameter-type-list)
or
D (⌈identifier-list⌋)

This declaration declares D to be a function that has parameters specified in the parameter type list and returns a value of type T1. The identifiers representing the function parameters are specified by the parameter type list. The identifiers that indicate the parameters and their types are set by the parameter type list. A macro defined in the stdarg.h header file converts a list specified by an ellipsis (,...) into a parameter. A function with no parameters has its parameter type list set to void.

3.5 Type Names

The type name is the name of a type that indicates the size of a function or object. Syntactically, the type name omits the identifier from the declaration for a function or an object.

Type name has this syntax:

```
type-name ::=
          specifier-qualifier-list⌈abstract-declarator⌋
     abstract-declarator ::=
           pointer
          |⌈pointer⌋ direct-abstract-declarator
          direct-abstract-declarator ::=
               ( abstract-declarator )
              |⌈direct-abstract-declarator⌋ [⌈constant-expression⌋ ]
              |⌈direct-abstract-declarator⌋ [⌈parameter-type-list⌋
```

Here are examples of type names.

• int	- Specifies the int type.
• int*	- Specifies a pointer to the int type.
• int*[3]	- Specifies an array (3 elements) whose elements are pointers to int.
• int(*)[3]	- Specifies a pointer to an array (3 elements) whose elements have type int.
• int *( )	- Specifies a function that returns the pointer to int and has no parameter specifications.
• int(*)(void)	- Specifies a pointer to a function that has no parameters and returns int.
• int(*const[ ])(unsigned int,...)	- Specifies an array (undefined number of elements) that has an unsigned int parameter and an unspecified number of other parameters.

3.6 typedef

typedef defines a synonym for the type specified by the identifier. The defined identifier becomes the typedef name.

typedef name has this syntax

```
typedef-name ::=
          identifier
```

The next example defines type_ident as the typedef name having the type (T1) specified by the declaration specifier T. Therefore, the type_ident identifier has type T1.

```
typedef T type_ident;
type_ident D;
```

In the next example, distance has type int, and metricp is a pointer to a function that has no parameter specifications and returns type int. The type of z is the specified structure. zp is a pointer to this structure. The object distance is compatible with other int objects.

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

In the next declaration, type t1, the type pointed to by tp1, and the struct s1 type are compatible. However, the struct s2 type, type t2, the type pointed to by tp2, and the int type are not compatible.

```
typedef struct s1{int x ;}t1, *tp1;
typedef struct s2{int x ;}t2, *tp2;
```

This example

```
typedef signed int t;
typedef int plain;
struct tag{
          unsigned t:4;
          const t:5;
          plain t:5;
};
```

declares typedef name t having signed int type, typedef name plain having type int, and a structure having three bit fields, one named t included in the range [0, 15], an unnamed const-qualified bit field that should include values in the range [–16, +15] (if it could be accessed), and one named r that includes value in the range [–16, +15]. The first two bit field declarations are different because unsigned is a type specifier (t becomes the name of a structure member) and const is a type qualifier (qualifies t that can be referenced as a typedef name). Within the internal scope, if this declaration follows these declarations, as in

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

the f function is declared with type “function returning signed int with one unnamed parameter with type pointer to function returning signed int with one unnamed parameter with type signed int.” The identifier t is declared as type long.

On the other hand, typedef names can be used to improve code readability. The three declarations of the signal function shown below specify the same type as the first without using the typedef name.

```
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 specifies the initial values for objects. Object initialization is performed by initializers.
Initializers have this syntax:

```
initializer ::=
        assignment-expression
      | {initializer-list}
      | {initializer-list, }
      initializer-list ::=
            initializer
          | initializer-list, initializer
```

The object type or array type of unknown size can be initialized. Initializers are specified only for the number of objects that are initialized in the initializer list.

All expressions in the initializers or initializer lists for objects having static storage duration and objects having aggregate or union type are specified by constant expressions.

Identifiers declared with block scope and identifiers having external or internal linkage cannot be initialized.

(1) Objects having static storage duration

When an arithmetic type object having static storage duration is not initialized, it is implicitly initialized to 0. Similarly, a pointer type object having static storage duration is initialized to a null pointer constant.

(2) Objects having automatic storage duration

The initial value of an object having automatic storage duration that was not initialized is not guaranteed. A structure or union object having automatic storage duration is initialized by an initializer list or primary expression having compatible type.

(3) Character array

A character array can be initialized by a character string literal. Similarly, successive characters in the character string literal initialize the elements in the array.

This example defines the s array object with no type qualifier and array object t. The elements in each array are initialized by a character string literal.

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

This example is identical to the initialization shown above.

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

This example defines the pointer p to an object having type char array whose members are initialized by a character string literal.

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

(4) Initializing aggregate and union objects

- Aggregates
  An aggregate type object is initialized by an initializer list written in increasing subscript or member order. The specified initializer-list enclosed by bracec.
  If there are fewer initializers in the list than the number of members in the aggregate, the remaining members are implicitly initialized in the same way as objects having static storage duration.
  An array of unknown size has its size determined by the number of elements in the array based on the number of initializers and does not become an incomplete type.

- Unions
  A union object is initialized by an initializer enclosed by braces for the first member of the union.

This example initializes an array x of unknown size to a one-dimensional array of type int having three members.

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

This example is a complete definition of initializers enclosed by braces. {1, 3, 5} initializes the first row of y[0][0], y[0][1], and y[0][2] in the y[0] array object list. Similarly, the next two rows initialize y[1] and y[2].

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

An array can be initialized by the following specification.

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

This example has the same result as the above example.

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

This example initializes the first column of z and sets the remaining elements to 0.

This example initializes a three-dimensional array.

q[0][0][0] is initialized to 1, q[1][0][0] to 2, and q[1][0][1] to 3, q[2][0][0] to 4, q[2][0][1] to 5, and q[2][1][0] to 6. The rest become 0.

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

This example initializes the three-dimensional array to the same values as above.

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

This example shows the fully bracketed form of the above initialization.

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

CHAPTER 4 TYPE CONVERSION

If there are two operands of different types in an expression, conversion is automatically performed. This is identical to the conversion obtained by using the cast operators. Automatic type conversion is called implicit type conversion. This chapter describes implicit type conversion.

Type conversion consists of usual conversion, truncation or rounding off conversion, or sign conversion. Table 4-1, “Type Conversions,” lists the type conversions.

Table 4-1. Type Conversions

After conversion / Before 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	+	×	○	○	○	○	○	○	○	±	±	±
	–	×	+	○	+	○	+	○	+	±	±	±
unsigned char		±	×	○	○	○	○	○	○	±	±	±
(signed) short int	+			×	○	×	○	○	○	±	±	±
	–			×	+	×	+	○	+	±		
unsigned short int				±	×	±	×	○	○			
(signed) int	+			×	○	×	○	○	○			
	–			×	+	×	+	○	+			
unsigned int				±	×	±	×	○	○			
(signed) long int	+							×	○			
	–							×	+			
unsigned long int								±	×			
float										×	○	○
double											×	○
long double												×

signed can be omitted. However, signed char or unsigned char is applied depending on the compilation conditions (options, etc.) only for type char.

○	:	Valid conversion
×	:	Cannot convert type
+	:	Invalid value results (regarded as an unsigned integer).
±	:	There is no change in the bit image, but when a positive value cannot be represented, a valid value does not result.
Space:		The overflow during conversion is discarded. Sometimes the sign changes depending on the type after conversion.

4.1 Arithmetic operands

(1) Characters and integers (integral promotion)

Objects of char, short int, and int bit fields, or their signed or unsigned versions, or the enumeration type are converted to the int type if they fall within the ranges that can be represented by the int type. If the object cannot be represented by the int type, the conversion is to the unsigned int type. These are called integral promotions. All other arithmetic types are not modified by integral promotion.

Integral promotion preserves the value including the sign. A char without a qualifier is treated as signed.

(2) Signed and unsigned integers

When converting an unsigned integer to a larger signed integer, the value of the unsigned integer does not change. When a signed or unsigned integer is demoted to a smaller signed integer, or when an unsigned integer is converted into a larger signed integer, values that cannot be represented are discarded.

The following cases are conversions from signed integers to unsigned integers.

Table 4-2. Conversion from Signed to Unsigned Integers

		unsigned	
		Small Value Range	Large Value Range
signed	+	/	o
	–	/	+

o : Correct conversion

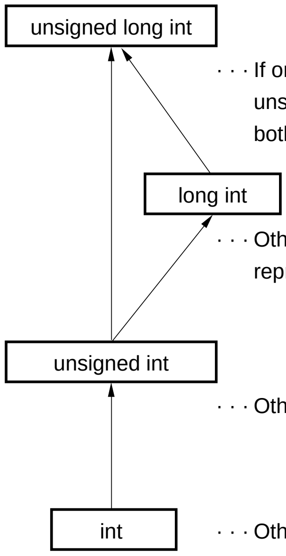
+ : Conversion to positive integers

/ : Reduced modulo (remainder) of the maximum value of the converted type plus one

(3) Usual arithmetic conversions

The type of the result of an arithmetic operation becomes a type having the wider range of values.

Figure 4-1. Usual Arithmetic Conversions



· · · If one operand is unsigned long int, the other operand is long int, and all of the values of unsigned int cannot be represented by long int when another operand is unsigned int, both operands are converted to unsigned long int.

· · · Otherwise, if one operand is long int and all of the operand values of the other can be represented by long int, conversion is to long int.

· · · Otherwise, if one operand is unsigned int, the other is converted to unsigned int.

· · · Otherwise, convert to the int type.

This compiler can be set to not intentionally convert to the int type based on the compilation conditions (optimization options). (For details, see Chapter 5, “Compiler Options,” in the CC78K3 C Compiler, Operation.)

4.2 Other operands

(1) lvalues and function designators

An lvalue is an expression (with an object type or an incomplete type other than void) that designates an object.

An lvalue that does not have an array type, incomplete type, or const-qualified type, or a structure or union that does not have const-qualified type members is a modifiable lvalue.

Except when the lvalue is the operand of the sizeof operator, the unary & operator, the ++ operator, the – – operator, the left operand of the . operator, or the assignment operator, the lvalue that does not have an array type is converted to the value stored in the designated object. Conversion does not make the lvalue disappear.

If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue. Otherwise, an lvalue that has incomplete type and not array type is not guaranteed.

Except for character arrays, an lvalue having type “array of type” is converted to an expression having type “pointer to type” that points to the first member of the array object. This is not an lvalue.

The function designator is an expression that has function type. Except for the operand of the sizeof operator or unary & operator, a function designator that has type “function returning type” is converted into an expression that has type “pointer to function returning type.”

(2) void

The (nonexistent) value of a void expression (an expression having type void) cannot be used in any way. Implicit or explicit conversions, except to void, cannot be applied to this expression. If an expression of another type occurs in a context requiring a void expression, its value or designator is discarded.

(3) Pointers

A pointer to void can be converted to a pointer to any incomplete type or object type. A pointer to any incomplete type or object type can be converted to a pointer to void. The result must be equal to the original pointer.

For any qualifier q, a pointer to a non-q-qualified type may be converted to a pointer to a q-qualified version of that type. Its value is stored at the original pointer. The converted pointer is equal to the original pointer.

An integral constant expression with value 0 or such an expression cast to the void * type is called a null pointer constant. If a null pointer constant is assigned to some pointer or is equal to some pointer, or is compared, the null pointer constant is converted into that pointer. The pointer called a null pointer is not guaranteed to be equal to the pointer to any object or function.

Two null pointers that are cast to type “pointer to type” become equal.

[MEMO]

CHAPTER 5 OPERATORS AND EXPRESSIONS

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

The C language offers a rich variety of operators for performing arithmetic and logical operations. In particular, the C language has operators that perform bit and address computations.

An expression is a sequence of operators and operands that specifies the computation of values, designates an object or a function, generates side effects, or is their combination.

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

char mark[SIZE+1];            + - Arithmetic operator

main()
{
    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;      + - Relational 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
         }
    }
    lprintf ("Total %d\n", count);
```

```
loop1:
       goto loop1;
}

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

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

Table 5-1, “Order of Operator Evaluation,” shows the operators used in the C language and their precedence.

Table 5-1. Order of Operator Evaluation

Type	Operator	Associativity	Precedence
Postfix	[] () . -> ++ – –	→	High
Unary	++ – – & * + – ~ ! sizeof	←	
Cast	(type-name)	←	
Multiplicative	* / %	→	
Additive	+ –	→	
Shift	<< >>	→	
Relational	< > <= >=	→	
Equality	== !=	→	
Bitwise AND	&	→	
Bitwise-exclusive OR	^	→	
Bitwise OR	|	→	
Logical AND	&&	→	
Logical OR	||	→	
Conditional	? . :	←	
Assignment	= *= /= %= += –= <<= >>= &= ^= |=	←	
Comma	,	→	Low

Operators in the same row have the same precedence. If an expression has two or more operators with the same precedence, evaluation is in the direction of the arrows shown under Associativity in the table.

5.1 Primary Expressions

The primary expressions are:

- Identifier declared as an object or Function
- Constant
- String literal
- Parenthesized expression

An identifier which becomes a primary expression is an lvalue for an object and a function designator for a function. The type of the constant is determined based on the value to be set as described in section 2.3, “Constants.” A string literal is an lvalue that has a type described in section 2.4, “Strings.”

5.2 Postfix Operators

A postfix operator is an operator placed after an object to indicate its name. Postfix operators have this syntax:

```
postfix-expression ::=
          primary-expression
         | postfix-expression [subscript]
         | postfix-expression (⌈argument-expression-list⌋)
         | postfix-expression . identifier
         | postfix-expression -> identifier
         | postfix-expression ++
         | postfix-expression – –
         argument-expression-list ::=
                assignment-expression
               | argument-expression-list, assignment-expression
```

(1) Array subscripting

Postfix operator	[ ] Subscript operator

[ ] Subscript operator

[Function]

The subscript operator [ ] specifies a member in the array object. The array E1[E2] defines the same element as (*(E1+(E2))). That is, the value of E1 is the pointer to the first member of the array. E2 is member E2 (counting from 0) in E1. For a multidimensional array, the number of dimensions in the array and the subscript operator are successive.

In this example, x becomes a 3-by-5 array of int. x has three member objects and each one is an array of five integer members.

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

By successively specifying the subscript operator, a multidimensional array can be specified. When E is an n-dimensional array of i x j x....x k (n ³ 2), E can be represented by n subscript operators. E becomes a pointer to the (n–1)-dimensional array of j x...x k.

[Syntax]

```
postfix-expression [subscript-expression]
```

[Constraints]

The postfix expression must be a “pointer to object type.” The subscript expression is specified by an integral type. The result has type “type.”

(2) Function call

Postfix operator	( ) Function call

( ) Function call

[Function]

A function is called. A function call is performed by the postfix operator ( ). The postfix expression denotes the called function and indicates the arguments passed to the called function enclosed by parentheses.

A called function without a storage class or type specification is interpreted as being an external object or calling a function that returns int without information related to the arguments. That is, this next declaration is performed implicitly.

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

In a function call, more efficient objects are generated by the function prototype declaration. A function prototype declaration specifies the value returned by the function, argument type, and storage class.

If a function prototype declaration is not referenced in a function call, each argument is integral promoted. This is called default integral promotion.

[Syntax]

```
postfix-expression (⌈argument-expression-list⌋)
```

[Constraints]

The called function is a function that returns void or an object except for an array. The postfix expression is a pointer to this function.

In a function call that includes a prototype, the argument type becomes a type that can be assigned to the corresponding parameter. The number of arguments must match.

(3) Structure and union members

Postfix operators	**. , ->**

<1> .Dot

[Function]

The dot (.) specifies a member object in a structure or union. The value of the postfix expression becomes the value of specified member.

[Syntax]

```
postfix-expression . identifier
```

<2> -> arrow

[Function]

The right arrow -> specifies the member object in a structure or union. The value of the postfix expression becomes the value of the specified member.

Postfix operators . , ->

Example using the dot (.), comma (,), and member-selection (->) operators

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

[Syntax]

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

(4) Postfix increment and decrement operators

Postfix operators	**++, – –**

<1> Postfix increment operator

[Function]

The postfix increment operator adds one to the value of the object. This operation considers the type of the object.

[Syntax]

```
postfix-expression ++
```

<2> Postfix decrement operator

[Function]

The postfix decrement operator subtracts one from the value of the object. This operation considers the type of the object.

[Syntax]

```
postfix-expression – –
```

[Constraints]

The operand of the postfix increment and postfix decrement operators is a modifiable lvalue that has qualified or unqualified scalar type.

5.3 Unary Operators

A unary operator operates on one object and item. The unary operators are:

- Prefix increment and decrement operators

  ++ – –
- Address and indirection operators

  & *
- Unary arithmetic operators

  + – ~ !
- sizeof operator

  sizeof

Unary operators have this syntax:

```
unary-expression ::=
            postfix-expression
          | ++ unary-expression
          | – – unary-expression
          | unary-operator cast-expression
          | sizeof unary-expression
          | sizeof ( type-name )
          unary-operator ::= one of
               & * + – ~ !
```

(1) Prefix increment and decrement operators

Unary operator	**++, – –**

<1> Prefix increment operator

[Function]

The prefix increment operator adds one to the value of the object. The expression ++E of the prefix increment operator has the same result as the following expressions.

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

[Syntax]

```
++ unary-expression
```

Unary operator **++, – –**

<2> Prefix decrement operator

[Function]

The prefix decrement operator subtracts one from the value of the object. The expression – –E of the prefix decrement operator has the same result as the following expressions.

```
E = E – 1
or
E –= 1
```

[Syntax]

```
– – unary-expression
```

[Constraint]

The operand of the prefix increment and prefix decrement operator is a modifiable lvalue that has qualified or unqualified scalar type.

(2) Address and indirection operators

Unary operators	**&, ***

<1> Unary & operator

[Function]

The address of the specified object is returned.

[Syntax]

```
& cast-expression
```

<2> Unary * operator

[Function]

The value pointed to by the specified pointer is returned.

[Syntax]

```
* cast-expression
```

[Constraints]

The operand of the unary & operator is an lvalue that points to an object that is not declared with the register storage-class specifier. A function designator or bit field cannot be used as the operand of the unary & operator.

The operand of the unary * operator has the pointer type.

(3) Unary arithmetic operators

Unary operators	**+, –, ~, !**

+, –, ~, ! operators

[Function]

The unary + operator performs positive integral promotion on the operand.

The unary – operator performs negative integral promotion on the operand.

The unary ~ operator returns the bitwise complement of the operand.

The unary ! operator is called the logical negation ! operator. The logical negation ! operator returns 1 when the operand is 0. Otherwise, it returns 0.

[Syntax]

```
+ cast-expression
– cast-expression
~ cast-expression
! cast-expression
```

[Constraints]

The operand of the unary + operator has scalar type.

The operand of the unary – operator has arithmetic type.

The operand of the unary ~ operator has integral type.

The operand of the unary ! operator has scalar type.

(4) sizeof operator

Unary operator	**sizeof operator**

sizeof operator

[Function]

The size of the specified object is returned in byte units. The return value is determined by the object type. The value of the object is not evaluated.

sizeof applied to an object of type char, unsigned char, or signed char (including qualified versions) returns one.

For an object of the array type, the result is the total number of bytes in the array. When the object has structure or union type, the result is the total number of bytes in the object including internal padding, that was inserted to guarantee the space.

The result is an integer constant of type size_t. This is defined in the stddef.h header file. The sizeof operator is primarily used in storage allocation and exchanges in the I/O system.

This example determines the number of members in an array by dividing the total number of bytes in the array by the size of a member.

```
sizeof array/sizeof array[0];
```

[Syntax]

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

[Constraints]

An expression having function type or incomplete type or an lvalue pointing to the bit field object cannot be used.

5.4 Cast Operators

The cast operator changes the type of the data. The cast operator is used for pointer type conversion.

Cast operator	**(type-name)**

Cast operator

[Function]

The object type converts to the type in the parentheses.

[Syntax]

```
(type-def) cast-expression
```

[Constraints]

The type name, except when void type is specified, has scalar type. The converted object file also has scalar type.

5.5 Arithmetic Operators

Arithmetic operators are divided into multiplicative operators and additive operators based on the precedence. A multiplicative operator determines the product, or the quotient and remainder between two operands. An additive operator determines the sum or difference between two operands.

The sum, difference, product, quotient, and remainder of two operands are represented by the binary + operator, binary - operator, binary * operator, binary / operator, and binary % operator, respectively.

• Multiplicative operators	*, /, %
• Additive operators	+, –

Both operands of the binary + operator have arithmetic type, or one is a pointer to an object and the other has integral type.

The operands of the binary – operator have the following types.

- Both have arithmetic types.
- Both are pointers to objects (including qualified and unqualified) having compatible types.
- The first operand is a pointer to an object and the second operand has integral type.

The operands of the binary * operator and the binary / operator have arithmetic type. The operands of the binary % operator have integral type.

Table 5-2. Multiplicative Operators

a/b		b	
		+	–
a	+	+	–
	–	–	+

a%b		b	
		+	–
a	+	+	+
	–	–	–

Division is performed on signed digits and the fractional part is discarded. Similarly, a multiplicative operation is performed on signed digits. The result of the % (modulo) operator assigns the signs shown in Table 5-2 to the value computed with the sign. Table 5-2 shows the computation result of the signs of only two operands.

(1) Multiplicative operators

Multiplicative operators	***, /, %**

<1> Binary * operator

[Function]

The * operator determines the product of two operands.

[Syntax]

```
multiplicative-expression * cast-expression
```

<2> Binary / operator

[Function]

The / operator determines the quotient when the first operand is divided by the second.

[Syntax]

```
multiplicative-expression / cast-expression
```

<3> Binary % operator

[Function]

The % operator determines the remainder when the first operand is divided by the second.

[Syntax]

```
multiplicative-expression % cast-expression
```

[Constraints]

Both operands of a multiplicative operator have arithmetic type.

(2) Additive operators

Additive operators	**+,-**

<1> Binary + operator

[Function]

This determines the sum of two operands.

[Syntax]

```
additive-expression + multiplicative-expression
```

<2> Binary - operator

[Function]

This determines the difference obtained by subtracting the second operand from the first.

[Syntax]

```
additive-expression - multiplicative-expression
```

[Constraints]

Both operands have arithmetic type, or one is a pointer to an object and the other has integral type. The operands for subtractions have the following types.

- Both have arithmetic type.
- Both are pointers to objects (qualified and unqualified) having compatible types.
- The first operand is a pointer to an object. The second object has integral type.

5.6 Shift Operators

A shift operator shifts the operand of the operator in the direction specified by the symbol. If the shifting operator has a signed type, an arithmetic shift is performed. If an unsigned type, a logical shift is performed.

The shift operator has this syntax:

```
shift-expression ::=
            additive-expression
          | shift-expression << additive-expression
          | shift-expression >> additive-expression
```

Table 5-3. Shift Operators

a<<b		b[Note 1]
a	+	0
	–	0

a>>b		b[Note 1]
a	+	0
	–	–1

Note 1 • The table shows cases where the digit exceeds the bit width or the result of the shift overflows.
• If a negative number is specified, it has unsigned type and processed as a positive number.

Shift operators <<, >>

<1> << operator

[Function]

The left operand is shifted left by the value in the 16 least-significant bits of the right operand. The vacated bits are filled with zeros.

If E1 has unsigned type in E1 << E2, the result is E1 multiplied by the square of E2. If E1 has unsigned long type, the result is reduced modulo ULONG_MAX+1 or UINT_MAX+1.

ULONG_MAX+1 and UINT_MAX+1 are defined in limits.h.

For information on limits.h, see Section 10.2 (8), “limits.h.”

[Syntax]

```
shift-expression << additive-expression
```

<2> >> operator

[Function]

The left operand is shifted to the right by value in the right operand. If E1 has unsigned type, a logical shift is performed. In a logical shift, vacated bits are filled with zeros after shifting.

When E1 has signed type, an arithmetic shift is performed. In an arithmetic shift, vacated bits are filled with the sign bit. The result is E1 divided by the square of E2.

[Syntax]

```
shift-expression >> additive-expression
```

[Constraints]

The operand to be shifted must have integral types.

5.7 Relational Operators

The operators that indicate relations are relational operators that indicate the relative sizes between two operands and equality operators that indicate whether they are equal.

In a relational operator, the size relationship when two pointers are compared is determined by the relative position in the address space of the objects being pointed to.

The relational operators have this syntax:

```
relational-expression ::=
            shift-expression
          | relational-expression < shift-expression
          | relational-expression > shift-expression
          | relational-expression <= shift-expression
          | relational-expression >= shift-expression
```

The equality operators have this syntax:

```
equality-expression ::=
            relational-expression
          | equality-expression == relational-expression
          | equality-expression != relational-expression
```

When comparing the sizes by relational operators, the following types hold.

- Both operands have arithmetic type.
- Both are pointers to objects (qualified and unqualified) having compatible types.
- Both are pointers to incomplete types having compatible types.
  When comparing by equality operators, the following types hold.
- Both operands have arithmetic type.
- Both are pointers to objects (qualified and unqualified) having compatible types.
- One is a pointer to an object type or incomplete type and the other is a pointer to void.
- One is a pointer and the other is a null pointer constant.

(1) Relational operators

Relational operators	**<, >, <=, >=**

<1> < operator

[Function]

If the first operand is less than the second operand, 1 is returned. Otherwise, 0 is returned.

[Syntax]

```
relational-expression < shift-expression
```

<2> > operator

[Function]

If the first operand is greater than the second operand, 1 is returned. Otherwise, 0 is returned.

[Syntax]

```
relational-expression > shift-expression
```

Relational operators <, >, <=, >=

<3> <= operator

[Function]

If the first operand is less than or equal to the second operand, 1 is returned. Otherwise, 0 is returned.

[Syntax]

```
relational-expression <= shift-expression
```

<4> >= operator

[Function]

If the first operand is greater than or equal to the second operand, 1 is returned. Otherwise, 0 is returned.

[Syntax]

```
relational-expression >= shift-expression
```

[Constraints]

When comparing the size, the following types hold.

- Both operands have arithmetic type.
- Both are pointers to objects (qualified and unqualified) having compatible types.
- Both are pointers to incomplete types having compatible types.

(2) Equality operators

Equality operators	**==, !=**

<1> == operator

[Function]

If the two operands are equal, 1 is returned. Otherwise, 0 is returned.

[Syntax]

```
equality-expression == relational-expression
```

<2> != operator

[Function]

If the two operands are not equal, 1 is returned. Otherwise, 0 is returned.

[Syntax]

```
equality-expression != relational-expression
```

[Constraints]

When comparing, the following types hold.

- Both operands have arithmetic type.
- Both are pointers to objects (qualified and unqualified) having compatible types.
- One is a pointer to an object type or incomplete type, and the other is a pointer to void.
- One is a pointer, and the other is a null pointer constant.

5.8 Bitwise Logical Operators

Bitwise logical operators perform logical operations in bit units on the object values. The bitwise logical operators are AND, exclusive OR, and OR, represented by &, ^, and |, respectively.

Bitwise logical operators have this syntax:

```
AND-expression ::=
            equality-expression
          | AND-expression & equality-expression
exclusive-OR-expression ::=
            AND-expression
          | exclusive-OR-expression ^ AND-expression
inclusive-OR-expression ::=
            exclusive-OR-expression
          | inclusive-OR-expression | exclusive-OR-expression
```

(1) Bitwise AND operator

Bitwise logical operator **&**

Binary & operator

[Function]

The bitwise AND operator & returns the bitwise logical product. The bitwise AND operator returns 1 if each corresponding bit is 1. The bitwise AND operator is specified by the binary & operator.

Table 5-4. Bitwise AND Operator

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

[Syntax]

```
AND-expression & equality-expression
```

[Constraints]

Both operands of the binary & operator have an integral type.

(2) Bitwise exclusive OR operator

Bitwise logical operator **^**

Binary ^ operator

[Function]

The exclusive OR operator ^ returns the bitwise logical sum. The bitwise exclusive OR operator returns 1 if the corresponding bits are different.

Table 5-5. Bitwise Exclusive OR Operator

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

[Syntax]

```
exclusive-OR-expression ^ AND-expression
```

[Constraints]

Both operands of the binary ^ operator have integral types.

(3) Bitwise OR operator

Bitwise OR operator **|**

Binary | operator

[Function]

The bitwise OR operator | returns the bitwise logical sum. The bitwise OR operator returns 0 if the corresponding bits are 0.

Table 5-6. Bitwise OR Operator

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

[Syntax]

```
inclusive-OR-expression ^ exclusive-OR-expression
```

[Constraints]

Both operands of the binary | operator have integral types.

5.9 Logical Operators

Logical operators find the logical product and logical sum of two operands. The logical product is specified by the logical AND operator &&. The logical sum is specified by the logical OR operator ||. Both operands of both logical operators have scalar type. A 0 or 1 integral type is returned.

Logical operators have this syntax:

```
logical-AND-expression ::=
        OR-expression
      | logical-AND-expression  && OR-expression
logical-OR-expression ::=
        logical-AND-expression
      | logical-OR-expression  || logical-AND-expression
```

(1) Logical AND operator

Logical operator	&&

Binary && operator

[Function]

The binary && operator takes logical AND of two operands. The logical AND returns 1 if both operands are not 0. Otherwise, it returns 0.

Table 5-7. Logical AND Operator

		Value of Left Operand	
		0	Not 0
Value of Right Operand	0	0	0
	Not 0	0	1

[Syntax]

```
logical-AND-expression && OR-expression
```

[Constraints]

Both operands of the binary && operator have scalar type.

The binary && operator evaluates the operands from left to right. If the value of the left operand is 0, the right operand is not evaluated.

(2) Logical OR operator

Logical operator	| |

Binary | | operator

[Function]

The binary | | operator takes the logical OR of two operands. The logical OR returns 0 if both operands are 0. Otherwise, it returns 1.

Table 5-8. Logical OR Operator

		Value of Left Operand	
		0	Not 0
Value of Right Operand	0	0	1
	Not 0	1	1

[Syntax]

```
logical-OR-expression || logical-AND-expression
```

[Constraints]

Both operands of the binary | | operator have scalar type.

The binary | | operator evaluates the operands from left to right. If the value of the left operand is not 0, the right operand is not evaluated.

5.10 Conditional Operator

Conditional operator determines the next process to perform based on the value of the first operand. The conditional operator is specified by ? and :.

The conditional operator evaluates the first operand and if the value is not 0, the second operand is evaluated. If the first operand is 0, the third operand is evaluated. The result of the conditional operator becomes the second or third operand.

The conditional operator has this syntax:

```
conditional-expression ::=
          logical-OR-expression
        | logical-OR-expression ? expression : conditional-expression
```

Conditional operator ? :

Conditional operator ?, :

[Function]

The conditional operator evaluates the first operand. If it is not 0, the second operand is evaluated. If it is 0, the third operand is evaluated. The value of the conditional operator becomes the value of the second or third operand.

[Syntax]

```
logical-OR-expression ? expression : conditional-expression
```

[Constraints]

The operands of the conditional operator have the following types.

- The first operand has scalar type.
  The second and third operands have one of the following types.
- Both operands have arithmetic type.
- Both have compatible structure or union types.
- Both have void type.
- Both are pointers to objects having a qualified or unqualified version of compatible types.
- One is a pointer, and the other is a null pointer constant.
- One is a pointer to an object or incomplete type, and the other is a pointer to void or qualified version of void.

5.11 Assignment Operators

The assignment operators are simple assignment that stores the right operand in the left object and compound assignment that stores the result of the operation on both operands in the left object.

The assignment expression has this syntax:

```
assignment-expression ::=
            conditional-expression
          | unary-expression assignment-operator assignment-expression
          assignment-operator ::= one of
                = *= /= %= += –= <<= >>= &= ^= |=
```

(1) Simple assignment

Simple assignment	=

Unary assignment operator =

[Function]

Simple assignment converts the right operand to the type of the left operand and stores it in the left object.

In the next example, the value of type int that is returned from the function by the type conversion of simple assignment is converted into type char. The overflow is discarded. When comparing to –1, conversion to the int type is repeated. If the c variable declared without a qualifier is regarded as unsigned char, the conversion result will not be negative. Therefore, the comparison to –1 is never equal. In this case, the c variable for full portability is declared with type int.

```
int f(void);

char c;
/*…*/((c=f()) == –1)/* …*/
```

[Syntax]

```
unary-expression = simple-assignment
```

[Constraints]

The operands of simple assignment has the following types.

- Both operands have arithmetic type.
- The left operand has qualified arithmetic type. The right operand has arithmetic type.
- Both are structures or unions having compatible types.
- The right operand has a structure or union type. The left operand is its qualified version.
- Both are pointers to compatible types.
- One is a pointer to an object or incomplete type. The other is a pointer to void.
- One is a pointer. The other is a null pointer constant.
- Both are pointers. The left operand is a pointer to the qualified version of the right operand.

(2) Compound assignment

Assignment operators	*= /= %= += –= <<= >>= &= ^= |=

Compound assignment

[Function]

The compound assignment operators store the result of an operation on the left and right operands in the left object. The stored value is converted into the type of the left operand.

The compound assignment E1 op= E2 is identical to the simple assignment expression E1 = E1 op (E2) except the lvalue E1 is only evaluated once.

The following compound assignment operations have identical results as the simple assignments 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
```

Assignment operators *= /= %= += –= <<= >>= &= ^= |=

[Syntax]

```
simple-expression *= assignment-expression
simple-expression /= assignment-expression
simple-expression %= assignment-expression
simple-expression += assignment-expression
simple-expression –= assignment-expression
simple-expression <<= assignment-expression
simple-expression >>= assignment-expression
simple-expression &= assignment-expression
simple-expression ^= assignment-expression
simple-expression |= assignment-expression
```

[Constraints]

The left operand of the += and –= operators is a pointer to an object. The right operand has integral type. Or the left operand has qualified or unqualified arithmetic type. The operands of the other operators have arithmetic type.

5.12 Comma Operator

Comma operator	**,**

Comma operator

[Function]

The comma operator evaluates the left operand as void. Then the operand on the right is evaluated and its value is returned.

The comma operator in this chapter cannot appear when the comma is used as a punctuator as indicated by the syntax (in a function argument list or initializer list).

In the next example, the value of the second argument passed to function f() is determined by using the comma operator. The value of the second argument becomes 5 by the comma operator.

```
f(a, (t=3,t+2), c)
```

[Syntax]

```
expression, assignment-expression
```

5.13 Constant Expressions

The constant expressions are integral constant expressions, arithmetic constant expressions, address constant expressions, and constant expressions in initializers. The evaluation of constant expressions does not occur during execution but almost always during compilation.

The following operators cannot be used in a constant expression, except when used in the sizeof operator.

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

(1) Integral constant expressions

An integral constant expression has integral type. The following can be used as operands in an integral constant expression.

- Integral constant
- Enumeration constant
- Character constant
- sizeof expression

Type conversion performed by a cast operator in an integral constant expression only converts from an arithmetic type to an integral type.

(2) Arithmetic constant expressions

An arithmetic constant expression has arithmetic type. The following can be used as operands in an arithmetic constant expression.

- Integral constant
- Enumeration constant
- Character constant
- sizeof expression

Type conversion by a cast operator in an arithmetic constant expression only converts from an arithmetic type to an arithmetic type, except when in a sizeof expression.

(3) Address constants

An address constant is a pointer to an object having a static storage duration or a pointer to a function designator. An address constant can be implicitly specified by using an expression of array or function type. When explicitly specified, the unary & operator is used.

An address constant can be specified by using the following operators. However, this cannot reference the value of the object.

- Array subscript [ ]
- . operator
- -> operator
- Address and unary operator
- Indirection * operator
- Pointer cast

(4) Constant expressions in initializers

The constant expression in an initializer is one of the following.

- Arithmetic constant expression
- Null pointer constant
- Address constant expression
- Address constant for an object where an integral constant was added or subtracted

CHAPTER 6 C LANGUAGE CONTROL STRUCTURES

This chapter describes the control structure of the C language and statements that are executed by C.

Generally, three basic control structures can be represented no matter how complex the processing. These three control structures are sequencing, selection, and iteration. In addition, jumps are used to forcibly alter the program flow.

The six statements executed in the C language are:

• Labeled statement	- Indicates the value of a switch statement and the branch destination of a goto statement.
• Compound statement (block)	- Set of multiple statements to be processed as one grammatical unit
• Expression statement	- Statement composed of one expression and a semicolon
• Selection statement	- Statement executed with the control expression of a selection processing structure
• Iteration statement	- Statement executed with the control expression having the loop structure
• Jump statement	- Controlled jump and its destination

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

        count = 0;
        for (i=0; i <= SIZE; i++)                          for -      Iteration statement
                 mark [i] = TRUE;
        for (i=0; i <= SIZE; i++)
                 if (mark[i]){                             if -       Selection statement
                             prime = i + i + 3;
                             lprintf("%d", prime);

                             if ((count%8)==0)putchar('\n');  if -  Selection statement
                             for (k=i+prime; k <= SIZE; k += prime)
                                    mark[k] = FALSE;
                 }
        }
        lprintf ("Total %d\n", count);

loop1:                                                     xxx: -     Labeled statement
        goto loop1;                                        goto -     Jump statement
}
```

(1) Sequential processing

A sequential process is executed from the top to the bottom in the order stipulated by the program. Sequentially executed statements are executed sequentially without any particular specification.

(2) Selective processing

Selective processing selects and executes the next statement to be executed based on the state of the executing program. The selection conditions are specified as control statements. A selection is made for execution from two statements or one that has many branches based on the control statement.

(3) Iterative processing

Iterative processing executes the same process multiple times. The statement being controlled is repeatedly executed for the state specified in the control statement or for the specified count.

(4) Jump processing

Jump processing forcibly leaves the current program flow and moves control to the specified label. Execution begins at the statement following the label name specified by the jump.

6.1 Labeled Statements

A labeled statement designates the jump destination of switch and goto statements. A switch statement selects and executes the statement specified by the control expression from the statement having multiple selections. The labeled statement becomes the label of the statement that is executed by the switch statement. The goto statement unconditionally branches to the corresponding label from the normal process flow.

A labeled statement has this syntax:

```
labeled-statement ::=
            identifier : statement
          | case constant-expression : statement
          | default : statement
```

(1) case

Labeled statement	case label

case label

[Function]

case is used only in a switch statement. The values for the control expression of the switch statements are enumerated.

[Syntax]

```
case constant-expression : 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;
{
```

[Description]

In this example, when f( ) returns the value 1, the first cast statement is selected and i=i+4 is executed. Similarly, when the value is 2, the second case is selected. And when 3, the third case is selected. The break statements in the example are placed in the switch statement to leave it.

The case statement is used when there are multiple selections.

(2) default

Labeled statement	**default label**

default label

[Function]

default is only used in the switch statement. default specifies the processing when there is no corresponding case in the switch statement.

[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;
}
```

[Description]

In the example, when 1 to 3 is the value returned by f( ), the corresponding case is selected and subsequent statements are executed. The break statements in the example are included to leave the switch statement. When a value other than 1 to 3 is returned by f( ), the statement following the default is executed and i is set to 1.

6.2 Compound Statement (Block)

A compound statement treats multiple statements as one grammatical unit. Multiple statements become a compound statement by enclosing them with braces { }. For example, when multiple processes will be performed in some state, those statements are enclosed by braces { } and executed.

A compound statement has this syntax.

```
        compound-statement ::=
                    {⌈declaration-list⌋⌈statement-list⌋}
              declaration-list ::=
                     declaration
                    | declaration-list declaration
              statement-list ::=
                     statement
                    | statement-list statement
```

6.3 Expression and Null Statements

A statement composed of one statement and a semicolon is called an expression statement. A statement composed of a semicolon is called a null statement. The null statement is used in a null loop body or to place a label.

The expression statement and null statement have this syntax.

```
⌈expression-statement⌋ :
                        ⌈expression⌋;
```

As shown in the following example, the function that is evaluated only to obtain the side effects as the expression statement can use the cast expression and explicitly discard the return value.

```
int p(int);
/*…*/
(void)p(0);
```

The null statement can be used as the loop body of the iteration statement.

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

This also can be used to place the label before the } that closes the compound statement.

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

6.4 Selection Statements

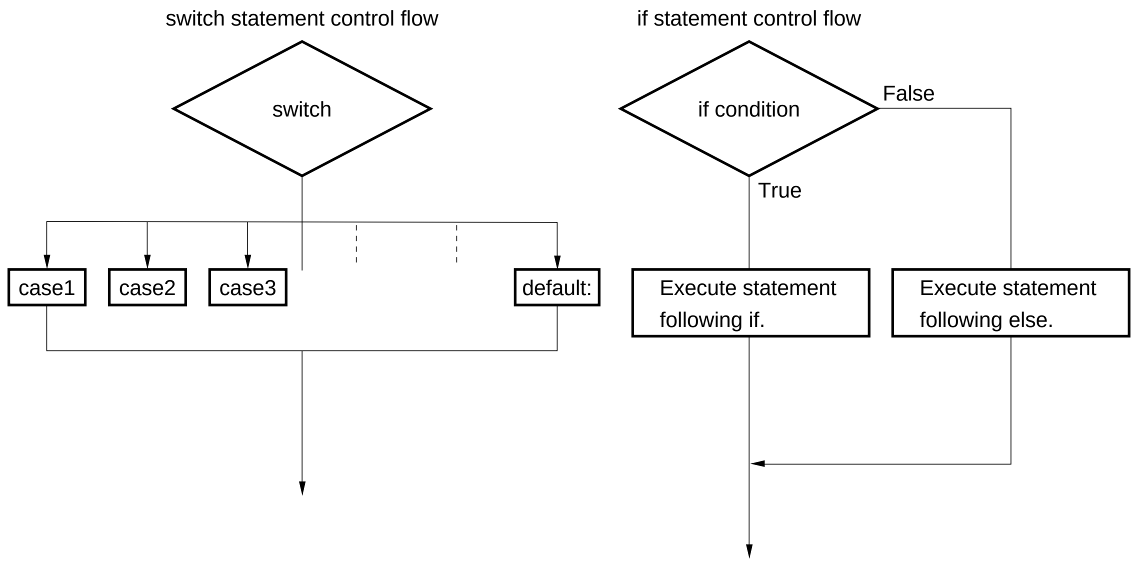
The selection statements are the if statement and switch statement. The selection statement selects the processing from the set of statements by using the value of the control expression enclosed by ( ).

The if and switch statements have this syntax.

```
selection-statement ::=
        if ( expression ) statement
      | if ( expression ) statement else statement
      | switch ( expression ) statement
```

The control flow of the if and switch statements are shown in the figure.

Figure 6-1. Selection Statement Control Flow


switch statement control flow
switch
case1
case2
case3
default:
if statement control flow
if condition
False
True
Execute statement following if.
Execute statement following else.

(1) if statement, if-else statement

Selection statement	if statement, if-else statement

[Function]

The if and if-else statements execute the next statement if the value of the control expression enclosed by ( ) is not 0. For the if-else statement, when the value of the expression is 0, the statement in the else statement is executed.

[Syntax]

```
if ( expression ) statement
if ( expression ) statement else statement
```

[Example]

```
if(i<10){
      /*111*/
}
else{
      /*222*/
}
```

[Description]

In this example, when the value of i is less than 10 in the control expression in the if statement, the {/*111*/} block is executed. If 10 or greater, the {/*222*/} block is executed.

[Constraints]

The control expression of the if statement has scalar type.

If the processing is not enclosed by braces { } after the if or if-else statement, only the processing in the statement after the if statement or else statement is considered to be the loop body.

(2) switch statement

Selection statement **switch statement**

[Function]

The switch statement moves control to the switch body of the case corresponding to the control expression enclosed by parentheses ( ). If there is no corresponding case statement for the control expression, the statement after default is executed. If there is no default, no statements are executed.

[Syntax]

```
switch ( expression ) 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;
}
```

[Description]

In the example, if the value returned by f( ) is 1, the first case is selected, and the i=i+4 expression is executed. Similarly, if the value is 2, the second case statement is selected. If the value is 3, the third case statement is selected. The break statements in the example are included to leave the switch statement.

[Constraints]

The constant expression has integral type. Each case expression is an integral constant expression.
Each case in one switch statement cannot be set to the same value. default can only be used once in one switch statement.

6.5 Iteration Statements

An iteration statement repeats execution of the loop body while the control statement enclosed by parentheses () is true (while not 0). The three iteration statements are:

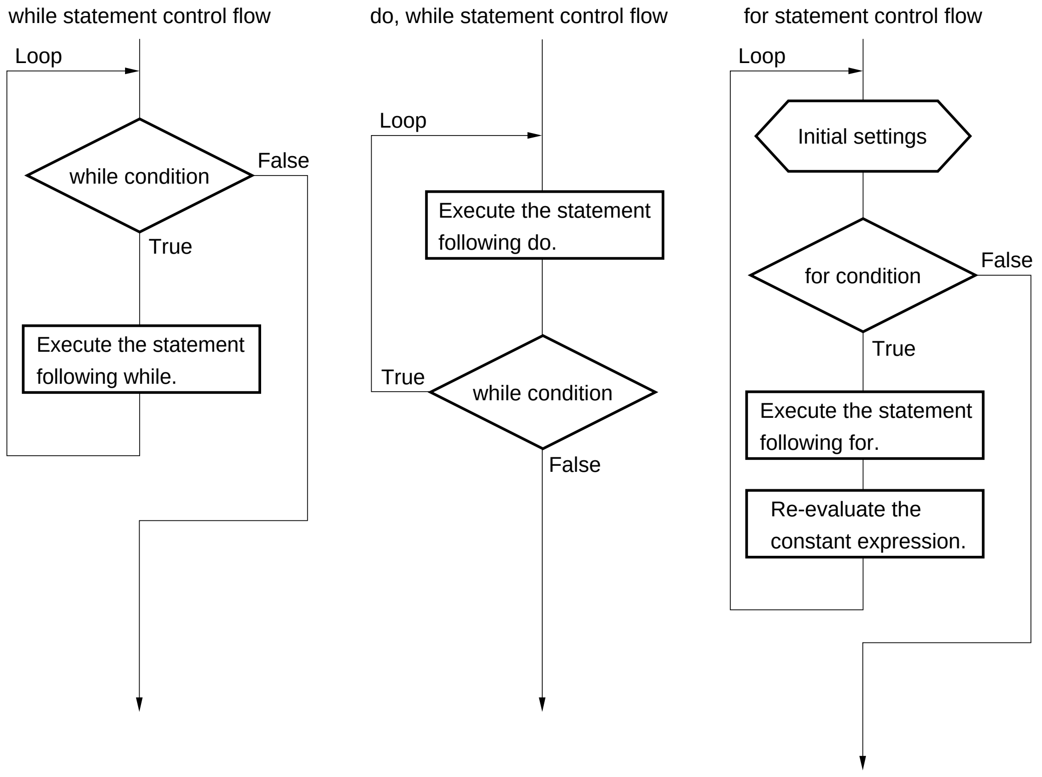
- while statement
- do statement
- for statement

The iteration statements have this syntax:

```
「iteration-statement」::=
            while ( expression ) statement
          | do statement while ( expression ) ;
          | for ( expression ; expression ; expression ) statement
```

The figure shows the control flow of the iteration statements.

Figure 6-2. Iteration Statement Control Flow


while statement control flow
Loop
while condition
False
True
Execute the statement following while.
do, while statement control flow
Loop
Execute the statement following do.
True
while condition
False
for statement control flow
Loop
Initial settings
for condition
False
True
Execute the statement following for.
Re-evaluate the constant expression.

(1) while statement

Iteration statement	**while statement**

[Function]

The while statement repeatedly executes the body of the loop while the constant expression enclosed by parentheses ( ) is true (while not 0). The while statement evaluates the constant expression before executing the loop body.

[Syntax]

```
while ( expression ) statement
```

[Example]

```
int i, x ;
i=1, x=0;,

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

[Description]

The example determines the sum of integers from 1 to 10 in x. The loop body of this while statement is the part enclosed by the braces { }. The control expression i < 11 returns 0 when i reaches 11. Therefore, the loop body is repeatedly executed while i from 1 to 10.

[Constraints]

The control expression has scalar type.

(2) do statement

Iteration statement	**do statement**

[Function]

The do statement repeatedly executes the loop body while the control expression enclosed by parentheses ( ) is true (while not 0). The do statement evaluates the control expression after executing the loop body.

[Syntax]

```
do statement while ( expression );
```

[Example]

```
int i, x;
i=1, x=0;

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

[Description]

The example determines the sum of the integers from 1 to 10 in x. The loop body of the do statement is the part enclosed by braces. The control expression i < 11 returns to 0 when i becomes 11. Therefore, the loop body is repeatedly executed while i is from 1 to 10.

[Constraints]

The control expression has scalar type.

(3) for statement

Iteration statement	for statement

[Function]

The for statement repeatedly executes the loop body while the second expression enclosed by parentheses ( ) is true (while not 0). The first expression initializes the variable used as the counter and is only executed once at the beginning of the loop. The decision about the counter is performed by the second expression. The third expression is executed last after each loop. After executing this expression, the variable is re-evaluated.

[Syntax]

```
for (「first-expression」;「second-expression」;「third-expression」) statement
```

[Example]

```
int i, x=0;

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

[Description]

The example determines the sum of the integers from 1 to 10 in x. The body of this for loop is x+=i. The control expression i<11 returns 0 when i reaches 11. Therefore, the loop body is repeatedly executed while i is from 1 to 10.

[Constraints]

The control expression has scalar type.
When no processing is enclosed by braces { } after the for statement, only the processing in the next line after the for statement is considered to be the loop body of the for statement.

6.6 Jump Statements

A jump statement leaves the current control flow and unconditionally moves control to any location. There are four jump statements

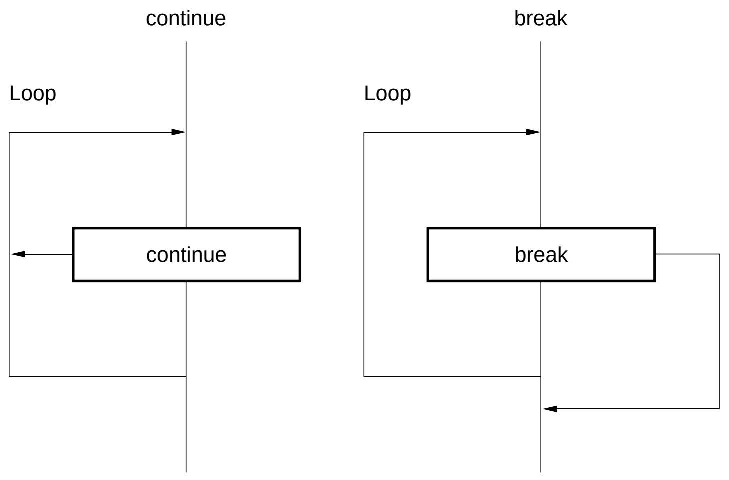
- goto statement
- continue statement
- break statement
- return statement

A jump statement has this syntax.

```
jump-statement ::=
            goto identifier ;
          | continue ;
          | break ;
          | return⌈expression⌋ ;
```

The figure shows the control flow of the jump statement.

Figure 6-3. Jump Statement Control Flow



(1) goto statement

Jump statement	goto statement

[Function]

The goto statement unconditionally jumps to the label name specified in the current function.

[Syntax]

```
goto identifier;
```

[Example]

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

[Description]

In this example, when control moves to the goto statement, the loop processing is leaved unconditionally and control moves to the statement following point.

[Constraints]

The jump destination (label name) indicated by the goto statement always points to any place in a function containing the goto statement.

(2) continue statement

Jump statement	continue statement

[Function]

The continue statement is used in the loop body of an iteration statement. Control flow by the continue statement unconditionally jumps to the end of the loop body. The continue statement is used in the iteration statement of the innermost side enclosing it.

[Syntax]

```
continue;
```

[Example]

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

[Description]

In this example, when the processing in the loop body reaches the continue statement, control unconditionally jumps to the contin label. The contin label does not have to indicate the jump destination. This example performs the same action as using the goto statement to replace the continue statement by goto contin.

[Constraints]

The continue statement can be the loop body or used only in the loop body.

(3) break statement

Jump statement	break statement

[Function]

The break statement moves control to the statement following the iteration statement or the switch statement to leave the iteration statement or switch statement.

[Syntax]

```
break;
```

[Example]

```
int f(void), i;

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

[Description]

This example uses the break statement in order to avoid performing unneeded evaluations in the switch statement. In the switch evaluation, when there is a corresponding case label, the switch statement is exited by the break statement.

[Constraints]

The break statement can be used as the switch body or only as the loop body.

(4) return statement

Jump statement	**return statement**

[Function]

The return statement leaves the function containing the return and moves control to the calling function. There can be multiple return statements in one function.

Closing the end of the function by the } is the same as executing a return statement that has no expression.

[Syntax]

```
return expression;
```

[Example]

```
void main()
{
         /*…*/
         int i;
         y=f(i);
         /*…*/
}

int f(int i)
{
         int x;
         /*…*/
         return(x);
}
```

Jump statement	**return statement**

[Description]

The f( ) function in this example returns to the main function when control moves to the return statement. Since the value of variable x is returned as the return value by the return statement, the value of variable x is assigned to variable y by the assignment operator.

[Constraints]

In a function returning void, an expression having a return value cannot be used in the return statement.

CHAPTER 7 STRUCTURES AND UNIONS

Structures and unions are sets of member objects having different types and assembled under one name. A structure assigns its member objects in contiguous space. A union assigns overlapping space.

7.1 Structures

A structure is a set of member objects contiguously allocated.

(1) Structure and structure variable declarations

For a structure, the structure declaration list and structure variables are declared by the struct keyword. Any name called the tag can be assigned to the structure declaration list. Then this tag can be used to declare structure variables having the same structure.

- Structure declaration

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

The next example declares the name, addr, and tell character arrays in the first struct having the tag data. no1 is declared as the variable. In the next struct, the no2, no3, no4, and no5 structure variables, which are identical to no1, are declared by using the tag.

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

(2) Structure declaration list

The structure declaration list describes the structure of the declared structure type. Each element in the structure declaration list is called a member. Storage is obtained in the order of the declared members.

The type of the member object is neither the incomplete type (array of unknown size) nor the function type. Consequently, a structure declaration list cannot be included in itself. Members can have any object type, except for the above types. Furthermore, a bit field that specifies the number of bits in the member can be specified.

When the variable value has the two values of 0 or 1, the bit field specifies one bit which is the minimum number of required bits. Multiple members can be stored in one integer space in the specification for the minimum number of required bits.

- Structure declaration list

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

- Bit fields

```
   unsigned int a:2;
   unsigned int b:3;
   unsigned int c:1;
```

(3) Arrays and pointers

A structure variable can be an array similar to other objects and can have a pointer. The elements in the array become structures in the structure array.

- Structure array

  The declaration of a structure array 1is performed in the same way as for other objects.

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

- Structure pointer

  A structure pointer has the features of the structure pointed to by the pointer. That is, when the pointer to the structure is incremented, the pointer is incremented by the structure size and points to the next structure.

In this example, dt_p is a pointer to the value of type struct data. When dt_p is incremented, it has the same value as &no[1].

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

(4) Referencing structure members

The two ways to reference structure members are to use the structure variables and to use the pointers to the variables. The . operator is used when referencing by using the structure variable. The -> operator is used when referencing by using a pointer.

- Referencing by a structure variable
  Referencing a member by using a structure variable uses the . operator.

```
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];
}
```

- Referencing by a pointer
  Referencing a member by using a pointer variable uses the -> operator.

```
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

A union is a collection of member objects that are allocated to the same space.

(1) Union and union variable declarations

For a union, the union keyword is used to declare a union list declaration and union variable. Any name called the tag can be assigned to the union declaration list. Then this tag can be used to declare union variables having the same structure.

- Union declaration

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

The example declares the name, addr, and tel character arrays in the first union that has the tag data. no1 is declared as its variable. Next, the no2, no3, no4, and no5 union variables, which have the same structure as no1, are declared by the tag of the union.

```
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 describes the structure of the declared union type. Each element in the union declaration list is called a member. The declared members use the same space.

The type of the member object is neither the incomplete type (array of unknown size) nor the function type. Consequently, a union declaration list cannot be included in itself. Members can have any object type, except for the above types.

- Union declaration list

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

(3) Arrays and pointers

As with other objects, union variables can be arrays and pointers.

- Union array

  A union array declaration is performed in the same way as for other objects.

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

- Union pointer

  A union pointer has the features of the union indicated by the pointer. That is, when a union pointer is incremented, the pointer is increased by the size of the union and points to the next union.

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

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

(4) Referencing union members

The two ways to reference union members are to use the union variables and the pointers to the variables. The . operator is used when referencing by using the union variable. The -> operator is used when referencing by using a pointer.

- Referencing by a union variable

  The . operator references a member by using a union variable.

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

- Referencing by a pointer

  The -> operator references a member by using a pointer variable.

```
union data{
        char name[12];
        char addr[50];
        char tel[12];
}*data_ptr;
data_ptr -> name[1] = 'N';
```

[MEMO]

CHAPTER 8 EXTERNAL DEFINITIONS

The program text after preprocessing becomes the translation unit. The program text of the translation unit is a column of external declarations. Since these are visible outside the function, they are called external.

An external definition defines functions or external objects. If identifiers declared with external linkage are used in expressions (except the operand of the sizeof operator), one external definition is required for the identifier somewhere in the program.

An external definition has this syntax.

```
translation-unit ::=
                external-declaration
               | translation-unit external-declaration
      external-declaration ::=
                function-declaration
                    | declaration
```

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

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

loop1:
    goto loop1;
}
```

8.1 Function Definitions

A function definition is an external definition. Even when the storage-class specifier is omitted, a function definition is considered to be defined by extern. An external function definition means that a defined function can be referenced from another file. For example, when some function in another file is used in a program composed of multiple files, this function has an external definition.

The storage-class specifier of a function is extern or static. When defined as extern, the function can be referenced from other functions. However, when defined as static, the function cannot be referenced from other files.

The external definition has this syntax.

function-definition ::=
⌈declaration-specifiers⌋⌈declarator declaration-list⌋ compound-statement

In the example, extern is the storage class specifier, and int is the type specifier. These can be omitted since they are the defaults. max(int a, int b) is the function declarator. {return a>b?a:b;} is the function body.

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

Since this function definition specifies the types of the parameters in the function declaration, type conversion is forcibly performed on the arguments. The function definition can be described based on the format of the identifier list for the parameter. The next example illustrates this.

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

A function address can be passed as an argument of a function call. The pointer to this function is created by using the function name in the expression.

```
int f(void);
g(f);
```

This example passes the f function to the g function by passing the pointer to the f function. The following is defined in the g function.

```
     g(int(*funcp)(void))
     {
            (*funcp)()/* or funcp()*/
     }
or
     g(int func(void))
     {
              func()/* or (*func)()*/
     }
```

8.2 External Object Definitions

An external object definition indicates that the identifier declaration for an object has file scope and an initializer. Also, an identifier declaration for an object having file scope that does not have a storage-class specifier nor an initializer, or a static storage-class specifier is a tentative definition. This definition has the file scope of an initializer equal to 0.

This list shows examples of external object definitions.

• 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;	- Tentative definition with external linkage
• static int i5;	- Tentative definition with internal linkage
• int i1;	- Valid tentative definition refers to previous one
• int i2;	- Linkage disagreement
• int i3;	- Valid tentative definition refers to previous one
• int i4;	- Valid tentative definition refers to previous one
• int i5;	- Linkage disagreement
• extern int i1;	- Refers to previous object with external linkage
• extern int i2;	- Refers to previous object with internal linkage
• extern int i3;	- Refers to previous object with external linkage
• extern int i4;	- Refers to previous object with external linkage
• extern int i5;	- Refers to previous object with internal linkage

CHAPTER 9 PREPROCESSING DIRECTIVES (COMPILER DIRECTIVES)

A preprocessing directive is a sequence of preprocessing tokens beginning with the # preprocessing token and ending with the new line character.

The only white space characters that can be used in a preprocessing token sequence are the space and the horizontal tab.

A preprocessing directive specifies the processing performed before compiling the source file. Preprocessing includes directives to conditionally process or skip a part of the source file, directives to include other source files, and directives to replace macros.

The preprocessing directive has this syntax.

```
preprocessing-file ::=
            ⌈group⌋
      group ::=
              group-part
            | group group-part
            group-part ::=
                  ⌈pp-tokens⌋ new-line
                 | if-section
                 | control-line
                 if-section ::=
                             if-group⌈elif-groups⌋⌈else-group⌋ endif-line
                       if-group ::=
                              #if constant-expression new-line⌈group⌋
                             | #ifdef identifier new-line⌈group⌋
                             | #ifndef identifier new-line⌈group⌋
                       elif-groups ::=
                              elif-group
                             | elif-groups elif-group
                             elif-group ::=
                                   #elif constant-expression new-line⌈group⌋
                             else-group ::=
                                   #else new-line⌈group⌋
                             endif-line ::=
                                   #endif new-line
                       control-line ::=
                              #include pp-tokens new-line
                             | #define identifier replacement-list new-line
                             | #define identifier lparen⌈identifier-list⌋) replacement-list new-line
                             | #undef identifier new-line
                             | #line pp-tokens new-line
                             | #error⌈pp-tokens⌋ new-line
                             | #pragma⌈pp-tokens⌋ new-line
                             | # new-line
```

```
lparen ::=
        ( without preceding white space
replacement-list ::=
        ⌈pp-tokens⌋
pp-tokens ::=
        preprocessing-token
      | pp-tokens preprocessing-token
new-line ::=
        new-line character
```

9.1 Conditional Compilation

Conditional compilation skips compiling a part of the source file based on the values of a constant expression. When the value of the constant expression defined by a conditional compilation directive is false, the next statement is not compiled. The sizeof operator, casts, and enumeration constants cannot be used in a constant expression.

#if, #elif, #ifdef, #ifndef, #else, and #endif are conditional compilation directives.

The following simple expressions can be specified in conditional compilation.

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

This simple expression returns 1 if the identifier is defined in the #define preprocessing directive. If the identifier is not defined or the definition has been undefined, 0 is returned.

(1) #if directive

Conditional compilation **#if**

[Function]

If the value of the constant expression is false, the compilation of a part of the source file is skipped.

[Syntax]

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

[Example]

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

[Description]

This example determines whether the next statement is compiled based on “FLAG == 0.”

If the value of FLAG is not 0, the program between the #if directive and #endif directive is not compiled and is compiled if 0.

(2) #elif directive

Conditional compilation **#elif**

[Function]

This directive occurs after an ordinary #if directive. If the constant expression of the #if directive is false, the constant expression following #elif is evaluated. If false, compiling the program after #elif is skipped.

[Syntax]

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

[Example]

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

[Description}

This example determines whether to compile the next statement depending on the value of FLAG. IF FLAG is 0, the program between #if to #endif is compiled. If not 0, the program between #elif and #endif is compiled.

(3) #ifdef directive

Conditional compilation **#ifdef**

[Function]

The #ifdef directive has the defined identifier as the constant expression of the #if directive.
If the identifier is defined by a #define directive, the rest of the program is compiled. If not defined or the declaration is removed, the compilation is skipped.

[Syntax]

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

[Example]

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

[Description]

Since ON is defined by the #define directive in the example, the program between #ifdef and #endif is compiled.
If ON is not defined, the program between #ifdef and #endif is not compiled.

(4) #ifndef directive

Conditional compilation **#ifndef**

[Function]

The #ifndef directive is identical to the !defined identifier as the constant expression of the #if directive. If this directive has been previously defined, the rest of the program is not compiled.

[Syntax]

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

[Example]

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

[Description]

Since ON is defined by the #define directive in the example, the program between #ifndef and #endif is not compiled. If ON is not defined, the program between #ifndef and #endif is compiled.

(5) #else directive

Conditional compilation **#else**

[Function]

For the #else directive, the rest of the program is compiled only when the identifier of the preceding conditional compilation directive is false. The directives before the #else directive are the #if, #elif, #ifdef, and #ifndef directives.

[Syntax]

```
#else new-line⌈group⌋
```

[Example]

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

[Description]

Since ON is defined by a #define directive in this example, the program between #ifdef and #else is compiled. If ON is not defined, the program between #else and #endif is compiled.

(6) #endif directive

Conditional compilation **#endif**

[Function]

The #endif directive indicates the end of the scope of the preceding conditional compilation directive.

[Syntax]

```
#endif new-line
```

[Example]

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

[Description]

#endif in the example indicates the end of the scope of #ifdef directive for conditional compilation.

9.2 Source File Inclusion

The #include preprocessing directive retrieves the specified header file and replaces the #include directive by the entire contents of the header file. There are three ways to include the source file by #include.

- #include <file-name>
- #include “file-name”
- #include preprocessing-tokens

***** preprocessing-tokens: character sequence defined by a #define directive

(1) #include < > directive

Source file inclusion	#include < >

[Function]

The specified header file is searched for in the directory specified in the environment variable, and the entire contents of the header file replace the #include directive.

[Syntax]

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

[Example]

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

[Description]

stdio.h is searched for in the directory specified by the environment variable. The #include <stdio.h> preprocessing directive is replaced by the contents of stdio.h.

(2) #include “ ” directive

Source file inclusion **#include “ ”**

[Function]

The source file included by this preprocessing directive is first searched for in the current directory. If the target file is not found, the directory specified by the environment variable is searched. The retrieved file replaces the #include directive.

[Syntax]

```
#include ”file-name” new-line
```

[Example]

```
#include ”myprog.h”
```

[Description]

myprog.h is searched for in the current directory or the directory specified by the environment variable. The preprocessing directive #include “myprog.h” is replaced by the contents of myprog.h.

(3) #include preprocessing token array directive

Source file inclusion	#include preprocessing tokens

[Function]

The header file is indicated by replacing the preprocessing tokens. The header file is searched for and replaces the #include directive.

[Syntax]

```
#include preprocessing-tokens new-line
```

[Example]

```
#define INCFILE “myprog.h”
#include INCFILE
```

[Description]

When including a source file by the #include preprocessing-tokens new-line, the specified preprocessing tokens must be replaced by <file-name> or “file-name” by macro replacement. When replaced by <file-name>, the source file is searched for in the directory specified by the environment variable. If there is a q character sequence, the current directory is searched. If not, the directory specified by the environment variable is searched.

9.3 Macro Replacement

Macro replacement replaces the character string (macro name) specified by the identifier with the replacement list. The macro name is replaced whenever it is specified. Macro replacement has the two formats of the object-like macro and function-like macro.

- Object-like macro
  #define identifier replacement-list new-line

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

(1) Argument substitution

Argument substitution is performed after the calling arguments of the function macro have been identified. If a # or ## preprocessing token is not placed before and a ## preprocessing token is not placed after the replacement list parameter, the macros in the list are expanded, and then all of them are replaced by the corresponding arguments.

(2) # operator

The # preprocessing token replaces the corresponding argument by the char character string processing token. If this is placed before the parameters in the replacement list, the corresponding arguments become a character or a character string.

(3) ## operator

The ## preprocessing token is linked to tokens in front and behind. Linking is performed before the next macro expansion, and the ## preprocessing token is deleted. If a macro name is in the generated token, the macro is expanded.

## operator example

```
#define debug(s,t) printf("x"#s"=%d,x"#t"=%s",x##s,x##t);
debug(1,2);
```

This is expanded to:

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

The char character string is linked to:

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

(4) Rescanning and further replacement

If there are macro names in a preprocessing token that underwent macro replacement and in preprocessing tokens in the rest of the source file, macro replacement is performed. The replaced macro is not replaced even when it is found while scanning a replacement list that does not include a preprocessing token sequence in the rest of the source file.

(5) Scope of macro definitions

A macro definition continues replacement until the corresponding #undef directive appears.

(i) #define directive

Macro replacement	#define

[Function]

The #define directive replaces the specified identifier by the replacement list. Identical identifiers after this directive are replaced by the replacement list.

[Syntax]

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

[Example]

```
#define PAI 3.1415
```

[Description]

This example replaces all PAIs that appear in the source file with 3.1415.

(ii) #define( ) directive

Macro replacement	**#define( )**

[Function]

A function-like macro directive replaces the identifier specified in the function format with the replacement list. Identical identifiers following this directive are replaced by the replacement list. Function-like macro replacement can handle replacements that include arguments.

[Syntax]

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

[Example]

```
#define F(n)(n*n)
int i;
i=F(2);
```

[Description]

F(2) in the example is replaced by (2*2) by the #define directive. Consequently, the value of i is 4.

A function-like macro is simple character replacement in contrast to a function definition. For safety, the replacement list in the #define directive is enclosed by parentheses ( ).

(iii) #undef directive

Macro replacement	#undef

[Function]

This ends the macro replacement directive.

[Syntax]

```
#undef identifier new-line
```

[Example]

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

[Description]

#undef in the example undefines the previously specified #define F(n)(n*n).

9.4 Line Control

Line control replaces the number of the line used while the compiler is compiling by the number specified by #line. If a character string is specified, the source file name in the compiler is replaced by the specified character string.

- If the line number is changed, this is specified:

```
#line digit-sequence new-line
```

- If the line number and file name are changed, this is specified:

```
#line digit-sequence ⌈character-string⌋ new-line
```

- Specifications other than the above are possible. In this case, the specified preprocessing tokens become one of the two described examples after all of the replacements.

```
#line preprocessing-tokens new-line
```

9.5 Error Directive

An error directive produces a message containing the specified preprocessing token sequence. This is specified by:

```
#error ⌈pp-tokens⌋ new-line
```

9.6 Pragma Directive

The #pragma directive tells the compiler about the specified character string. This C compiler uses the #pragma directive to produce 78K/0 code.

For details on the pragma directive, see **CHAPTER 11,** “**EXTENDED FUNCTIONS**.”

9.7 Null Directive

The null directive has no effect on the compiler.

```
# new-line
```

9.8 ASM Directive

The ASM directive includes the following lines as assembler source in the assembler source file module file output by the C compiler.

#asm indicates the beginning and #endasm indicates the end of the assembler source in the ASM directive.

[Syntax]

```
#asm
  :
#endasm
```

[Example]

```
#asm
       callf !init
#endasm
```

[Description]

In this example, callf !init between #asm and #endasm are not compiled, but are written to the assembler source module file output by the compiler.

9.9 Compiler Definition Macro Names

This compiler predefines the following macro names. These macro names cannot be changed during translation. Except for _ _LINE_ _ and _ _FILE_ _, the #define or #undef preprocessing directive cannot be applied.

_ _LINE_ _	Line number in current source (decimal number)
_ _FILE_ _	Source file name (character string literal)
_ _DATE_ _	Translation date of the source file (character string literal with the format of Mmmddyyyy)
_ _TIME_ _	Translation time of the source file (character string literal in the format of hh:mm:ss)
_ _STDC_ _	Decimal constant 1

In addition to the above, macro names indicating the series names of a device and ones indicating the device name are provided based on the target development device of the application product. These are specified based on the compilation options or device type in the C source in order to output object code for the target device.

- Macro name indicating the device's series name

  _ _KOS_ _

- Macro name indicating the device name

  A name for a device type name is preceded by _ _ and followed by _

  (Example) _ _9024_

* The device type name is identical to the one specified by the -C option.
  See data about device files for the device type names.

CHAPTER 10 LIBRARY FUNCTIONS

The C language does not have instructions to perform input and output (I/O) with external (peripheral) devices and equipment. The designers of the C language employed this design to keep the functions in the C language to a minimum. However, I/O operations are required in actual system development. Therefore, library functions are provided to perform I/O in the C language.

This C compiler has library functions for I/O, character manipulation, memory manipulation, program control, and mathematical functions.

10.1 Interface

A library function is used by issuing a function call. The call instruction performs function calls. In the case of the normal model, arguments are passed via a stack, or when possible, via a register, and return values are passed via a register. In the case of the static model, all arguments and return values are passed via a register.

10.1.1 Arguments

The calling side pushes arguments on and pops them off the stack. The called side only references that value.

The arguments are pushed on the stack in order from the end to the beginning.

The smallest unit pushed on the stack is 16 bits. A type larger than 16 bits is placed in 16-bit units in order from the high-order position. An 8-bit type is expanded to 16 bits.

The table lists the passing of the arguments or return values.

Table 10-1. Argument Passing List (Normal Model)

Argument Type	First Argument	Second Argument
1-byte, 2-byte integer	AX	Pass on stack
3-byte integer	AX, BC	Pass on stack
4-byte integer	AX, BC	Pass on stack
Floating-point number (float type)	AX, BC	Pass on stack
Other	Pass on stack	Pass on stack

In the static model, all arguments are passed via a register. Up to 3 arguments, or a total of 6 bytes, can be passed. Structure arguments passing is not supported in the static model.

Table 10-2. Argurment Passing List (Static Model)

Argument Type	First Argument	Second Argument	Third Argument
1-byte integer	A	B	H
2-byte integer	AX	BC	HL
4-byte integer*	AX, BC		
Floating-point number			
Other			

* In the case of 4-byte integer arguments, they are allocated to AX and BC, and remaining arguments (up to 2 bytes) are allocated to the HL or H registers.

10.1.2 Return value

A return value is stored in 16-bit units from the most-significant bits in registers BC to DE with a minimum unit of 16 bits. If a structure is returned, the first address of a structure is stored in BC. If a pointer is returned, it is stored in BC.

Table 10-3. Return Value Storage List (Normal Model)

Return Value Type	Storage Method
1-bit	CY
1-byte, 2-byte integer	BC
4-byte integer	BC (low-order), DE (high-order)
Floating-point number (float type)	BC (low-order), DE (high-order)
Structure	The structure to be returned is copied to a function-specific space. The address is stored in BC.
Pointer	BC

Table 10-4. Return Value Storage List (Static Model)

Return Value Type	Storage Method
1-bit	CY
1-byte	A
2-byte integer	AX
4-byte integer	AX (low-order), BC (high-order)
Floating-point number	
Structure	
Pointer	AX

* Storage method for the floating-point number and structure are not supported.

10.1.3 Saving the registers used by each library

A library which uses the following registers saves the register to be used on the stack

: HL register for the normal model

: DE register for the static model

A library that uses the saddr area saves the saddr area being used on the stack.

The work area used by the library uses the stack area.

This example shows the passing procedure for the arguments and the return value.

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

- For normal model

<1> Push the arguments on the stack (calling side).
The most-significant 16 bits of c and b, and the least-significant 16 bits of b are pushed in order on the stack.
a is passed by the ax register.

<2> Call func by using the call instruction (calling side).
The return address is pushed after the least-significant 16 bits of b, and control moves to the func function.

<3> Save the register used in the function (called side).
When HL is used, HL is pushed on the stack.

<4> Push the first argument passed in the register on the stack.

<5> Perform func function processing and store the return value in the register (called side).
The least-significant 16 bits of the long return value is stored in BC and the most-significant 16 bits in DE.

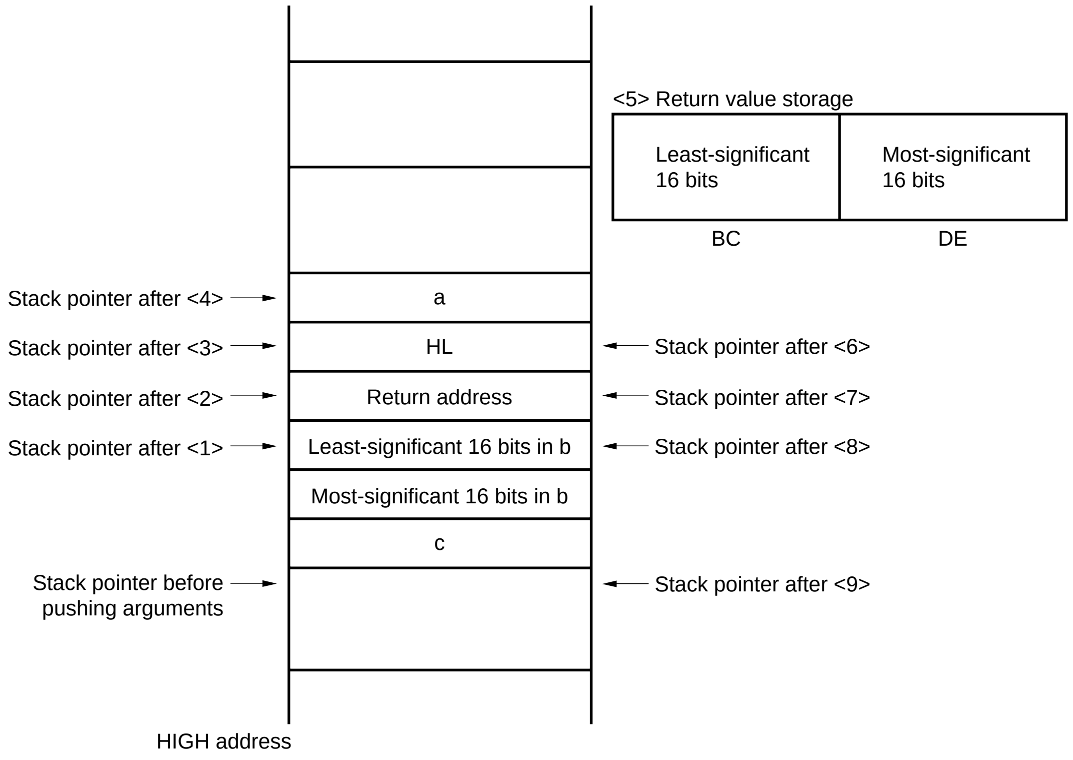
<6> Restore the first stored argument (called side).

<7> Restore the saved register (called side).

<8> Use the ret instruction to return control to the function called (called side).

<9> Remove an argument from the stack (calling side).
The number of bytes (2 byte units) in the argument is added to the stack pointer.
Six is added in the example in Figure 10-1.

Figure 10-1. Stack Area During a Function Call



10.2 Header Files

This C compiler has 13 header files that define or declare each library function, type, and macro.

However, the function defined in each header file differs depending on the presence/absence of model specification (-SM) or -ZI/-ZL options. For details, refer to Table 10-5, Standard Library Functions.

The header files for this C compiler are:

```
ctype.h      setjmp.h     stdarg.h      stdio.h
stdlib.h     string.h     error.h       errno.h
limits.h     stddef.h     math.h        float.h       assert.h
```

(1) ctype.h

ctype.h defines character and string functions. The following functions are defined in ctype.h.

However, if the -ZA compiler option (option that disables non-ANSI functions and enables of portion of ANSI functions) is specified, _toupper and _tolower are not defined. Instead, tolow and toup are defined. If -ZA is not specified, tolow and toup are not defined. In the static model, the following functions are not supported when specifying the -ZI (int and short are treated as char) option.

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

(2) setjmp.h

setjmp.h defines program control functions. setjmp and longjmp are defined in setjmp.h.
However, in the case of the static model, the following functions are not supported when specifying the -ZI option.
The following object is declare in setjmp.h.

- Declaration of jmp_buf, an int type array of size 11 (Normal Model)

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

- Declaration of jmp_buf, an int type array of size 2 (Static Model)

```
typedef int jmp_buf[2];
```

(3) stdarg.h

stdarg.h defines special functions. These functions are defined in stdarg.h.
The following function supports only normal model.

```
va_start     va_arg       va_end
```

This object is defined in stdarg.h. (only normal model)

- Declaration of va_list of type pointer to char

```
typedef char *va_list;
```

(4) stdio.h

stdio.h defines the I/O functions. These functions are defined in stdio.h.

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

These macros are declared.

```
#define      EOF          –1
#define      NULL         OL
```

(5) stdlib.h

stdlib.h defines character and string functions, memory functions, program control functions, mathematical functions, and special functions. The following functions are defined in stdlib.h.

However, if the -ZA compiler option (option that disables non-ANSI functions and enables of portion of ANSI functions) is specified, brk, sbrk, itoa, ltoa, and ultoa are not defined. Instead, strbrk, strsbrk, stritoa, strltoa, and strultoa are defined. If -ZA is not specified, strbrk, strsbrk, stritoa, strltoa, and strultoa are not defined.

```
atoi     atol      strtol    strtoul   calloc    free      malloc    realloc
abort    atexit    exit      abs       div       labs      ldiv      brk
sbrk     atof      strtod    itoa      ltoa      ultoa     rand      srand
bsearch  qsort     strbrk    strsbrk   stritoa   strltoa   strultoa
```

These objects are declared in stdlib.h.

- Declaration of div_t structure with int members quot and rem

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

- Declaration of ldiv_t structure with long int members quot and rem

```
typedef struct{
       long int quot;
       long int rem;
}ldiv_t;
```

- Definition of RAND_MAX macro

```
#define RAND_MAX 32767
```

- Declarations of macros

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

(6) string.h

string.h defines character and string functions, memory functions, and special functions. These functions are defined in string.h.

```
memcpy   memmove   strcpy    strncpy   strcat    strncat   memcmp
strcmp   strncmp   memchr    strchr    strcspn   strpbrk   strrchr
strspn   strstr    strtok    memset    strerror  strlen
strcoll  strxfrm
```

(7) error.h

error.h includes errno.h.

(8) errno.h

These objects are defined.

- Definitions of the EDOM, ERANGE, and ENOMEM macros

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

- Declaration of the errno external variable of type volatile int

```
extern volatile int errno;
```

(9) limits.h

These macros are defined in limits.h.

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

#define SCHAR_MAX               +127
#define SCHAR_MIN               –128
#define SHRT_MAX              +32767
#define SHRT_MIN              –32768
#define UCHAR_MAX               255U
#define UINT_MAX              65535U
#define ULONG_MAX        4294967295U
#define USHRT_MAX             65535U
#define SINT_MAX              +32767
#define SINT_MIN              -32768
#define SSHRT_MAX             +32767
#define SSHRT_MIN             -32768
```

However, when the -QU option is specified so that unqualified char is regarded as unsigned char, CHAR_MAX and CHAR_MIN are declared as shown below by the _ _CHAR_UNSIGNED_ _ macro declared by the compiler.

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

Moreover, when compiler option -ZI (int/short and unsigned int/unsigned short are treated as char type and unsigned char type, respectively) is specified, INT_MAX, INT_MIN, SHRT_MAX, SHRT_MIN, SINT_MAX, SINT_MIN, SSHRT_MAX, and SSHRT_MIN 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_MAX
```

When compiler option -ZL (long type and unsigned long type are treated as int type and unsigned int type, respectively) is specified, 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

These objects are declared and defined in stddef.h.

- Type declaration of int type ptrdiff_t

```
typedef int ptrdiff_t;
```

- Type declaration of unsigned int type size_t

```
typedef unsigned int size_t;
```

- Definition of NULL macro

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

- Definition of offsetof macro

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

* offsetof(type, member-designator)
This expands to an integral constant expression having type size_t. Its value is the offset in bytes from the beginning of the structure (designated by type) to the structure member (designated by the member designator). If the member designator is a declaration like “static type t;”, the result of evaluating the expression &(t.member-designator) must be an address constant. If the designated member is a bit field, the behavior is undefined.

(11) math.h

The following function is defined in math.h. (normal model only)

```
matherr
```

The following object is defined.

- Definition of HUGE_VAL macro

```
#define HUGE_VAL _HUGE
```

(12) float.h

The following objects are defined in float.h.

The macros are divided into the macros declared by the compiler when the size of the double type is 32 bits and macros defined by _ _DOUBLE_IS_32BITS_ _.

```
#ifndef _FLOAT_H

#if defined (_ _K0_ _)|| defined (_ _K0S_ _)
#ifndef _ _STATIC_MODEL_ _

#define FLT_ROUNDS       1
#define FLT_RADIX        2

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

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

#define FLT_MIN_EXP      –125
#define DBL_MIN_EXP      –125
#define LDBL_MIN_EXP     –125

#define FLT_MIN_10_EXP   –37
#define DBL_MIN_10_EXP   –37
#define LDBL_MIN_10_EXP  –37

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

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

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

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

#define FLT_MIN          1.17549435E-38F
#define DBL_MIN          1.17549435E-38F
#define LDBL_MIN         1.17549435E-38F

#else /*_ _DOUBLE_IS_32BITS_ _*/
#define FLT_MANT_DIG     24
#define DBL_MANT_DIG     53
#define LDBL_MANT_DIG    53

#define FLT_DIG          6
#define DBL_DIG          15
#define LDBL_DIG         15
```

```
#define FLT_MIN_EXP      –125
#define DBL_MIN_EXP      –1021
#define LDBL_MIN_EXP     –1021

#define FLT_MIN_10_EXP   -37
#define DBL_MIN_10_EXP   -307
#define LDBL_MIN_10_EXP  -307

#define FLT_MAX_EXP      +128
#define DBL_MAX_EXP      +1024
#define LDBL_MAX_EXP     +1024

#define FLT_MAX_10_EXP   +38
#define DBL_MAX_10_EXP   +308
#define LDBL_MAX_10_EXP  +308

#define FLT_MAX          3.40282347E+38F
#define DBL_MAX          1.7976931348623157E+308
#define LDBL_MAX         1.7976931348623157E+308

#define FLT_EPSILON      1.19209290E-07F
#define DBL_EPSILON      2.2204460492503131E-016
#define LDBL_EPSILON     2.2204460492503131E-016

#define FLT_MIN          1.17549435E-38F
#define DBL_MIN          2.225073858507201E-308
#define LDBL_MIN         2.225073858507201E-308
#endif /*_ _DOUBLE_IS_32BITS_ _*/

#endif /* !_ _STATIC_MODEL_ _*/
#endif /* _ _K0_ _||_ _K0S_ _*/

#define _FLOAT_H
#endif /*!_FLOAT_H*/
```

(13) assert.h

The following object is defined in assert.h. (normal mode only)

```
#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*/
```

10.3 Error Checking

The compiler calls some standard library for error checking and generates objects based on the option specification. The error checking contents are described next.

Error checking contents

Stack overflow:	Checks whether stack usage exceeds the stack area. An error results if stack usage exceeds the stack.

If an error is discovered during error checking, an error processing function in the run-time library is called. The error processing function is errstk.

10.4 Standard Library Functions

The standard library functions in this C compiler are classified and described based on their functions.

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

If there is error checking, the function name in the standard library is prefixed by “_@”. (Error checking is performed by specifying the error check option -L [S].)

The standard library functions are listed below.

○:	Function supported
×:	Function not supported
Δ:	Function supported but with restrictions

Table 10-5. Standard Library Function Name List (1/3)

Section	Function Name	Normal Model				Static Model				Error Check	Header File
		Without zi Without zi	With zi Without zi	Without zi With zi	With zi With zi	Without zi Without zi	With zi Without zi	Without zi With zi	With zi With zi		
1-01	isalnum	○	○	○	○	○	×	○	×	None	ctype.h
1-02	isalpha	○	○	○	○	○	×	○	×	None	
1-03	iscntrl	○	○	○	○	○	×	○	×	None	
1-04	isdigit	○	○	○	○	○	×	○	×	None	
1-05	isgraph	○	○	○	○	○	×	○	×	None	
1-06	islower	○	○	○	○	○	×	○	×	None	
1-07	isprint	○	○	○	○	○	×	○	×	None	
1-08	ispunct	○	○	○	○	○	×	○	×	None	
1-09	isspace	○	○	○	○	○	×	○	×	None	
1-10	isupper	○	○	○	○	○	×	○	×	None	
1-11	isxdigit	○	○	○	○	○	×	○	×	None	
1-12	tolower	○	○	○	○	○	×	○	×	None	
1-13	toupper	○	○	○	○	○	×	○	×	None	
1-14	isascii	○	○	○	○	○	×	○	×	None	
1-15	toascii	○	○	○	○	○	×	○	×	None	
1-16	_tolower	○	○	○	○	○	×	○	×	None	
1-17	_toupper	○	○	○	○	○	×	○	×	None	
1-18	tolow	○	○	○	○	○	×	○	×	None	
1-19	toup	○	○	○	○	○	×	○	×	None	
2-01	setjmp	○	○	○	○	○	×	○	×	None	setjmp.h
2-02	longjmp	○	○	○	○	○	×	○	×	None	
3-01	va_arg	○	○	○	○	×	×	×	×	None	stdarg.h
3-02	va_start	Δ	Δ	Δ	Δ	×	×	×	×	None	
3-03	va_end	○	○	○	○	×	×	×	×	None	
4-01	sprintf	○	×	○	×	×	×	×	×	Stack Over-flow	stdio.h
4-02	sscanf	○	×	○	×	×	×	×	×		
4-03	printf	○	×	○	×	×	×	×	×		
4-04	scanf	○	×	○	×	×	×	×	×		
4-05	vprintf	○	×	○	×	×	×	×	×		
4-06	vsprintf	○	×	○	×	×	×	×	×		
4-07	getchar	○	○	○	○	○	×	○	×	None	
4-08	gets	○	○	○	○	○	○	○	○	None	
4-09	putchar	○	○	○	○	○	×	○	×	None	
4-10	puts	○	○	○	○	○	×	○	×	None	

Table 10-5. Standard Library Function Name List (2/3)

Section	Function Name	Normal Model				Static Model				Error Check	Header File
		Without zi Without zi	With zi Without zi	Without zi With zi	With zi With zi	Without zi Without zi	With zi Without zi	Without zi With zi	With zi With zi		
5-01	atoi	○	×	○	×	○	×	○	×	None	stdlib.h
5-02	atol	○	○	×	×	×	×	×	×	None	
5-03	strtol	○	○	×	×	×	×	×	×	None	
5-04	strtoul	○	○	×	×	×	×	×	×	None	
5-05	calloc	○	○	○	○	○	×	○	×	None	
5-06	free	○	○	○	○	○	×	○	×	None	
5-07	malloc	○	○	○	○	○	×	○	×	None	
5-08	realloc	○	○	○	○	○	×	○	×	None	
5-09	abort	○	○	○	○	○	○	○	○	None	
5-10	atexit	○	○	○	○	○	×	○	×	None	
5-11	exit	○	○	○	○	○	×	○	×	None	
5-12	abs	○	×	○	×	○	×	○	×	None	
5-13	div	○	○	○	○	×	×	×	×	None	
5-14	labs	○	○	×	×	×	×	×	×	None	
5-15	ldiv	○	○	×	×	×	×	×	×	None	
5-16	brk	○	○	○	○	○	×	○	×	None	
5-17	sbrk	○	○	○	○	○	×	○	×	None	
5-20	itoa	○	○	○	○	○	×	○	×	None	
5-21	ltoa	○	○	×	×	×	×	×	×	None	
5-22	ultoa	○	○	×	×	×	×	×	×	None	
5-23	rand	○	×	○	×	○	×	○	×	None	
5-24	srand	○	○	○	○	○	×	○	×	None	
5-25	bsearch	○	○	○	○	×	×	×	×	None	
5-26	qsort	○	○	○	○	×	×	×	×	None	
5-27	strbrk	○	○	○	○	○	×	○	×	None	
5-28	strsbrk	○	○	○	○	○	×	○	×	None	
5-29	stritoa	○	○	○	○	○	×	○	×	None	
5-30	strltoa	○	○	×	×	×	×	×	×	None	
5-31	strultoa	○	○	×	×	×	×	×	×	None	

Table 10-5. Standard Library Function Name List (3/3)

Section	Function Name	Normal Model				Static Model				Error Check	Header File
		Without zi Without zi	With zi Without zi	Without zi With zi	With zi With zi	Without zi Without zi	With zi Without zi	Without zi With zi	With zi With zi		
6-01	memcpy	○	○	○	○	○	×	○	×	None	string.h
6-02	memmove	○	○	○	○	○	×	○	×	None	
6-03	strcpy	○	○	○	○	○	○	○	○	None	
6-04	strncpy	○	○	○	○	○	×	○	×	None	
6-05	strcat	○	○	○	○	○	○	○	○	None	
6-06	strncat	○	○	○	○	○	×	○	×	None	
6-07	memcmp	○	×	○	×	○	×	○	×	None	
6-08	strcmp	○	×	○	×	○	×	○	×	None	
6-09	strncmp	○	×	○	×	○	×	○	×	None	
6-10	memchr	○	○	○	○	○	×	○	×	None	
6-11	strchr	○	○	○	○	○	×	○	×	None	
6-12	strcspn	○	×	○	×	○	×	○	×	None	
6-13	strpbrk	○	○	○	○	○	○	○	○	None	
6-14	strrchr	○	○	○	○	○	×	○	×	None	
6-15	strspn	○	×	○	×	○	×	○	×	None	
6-16	strstr	○	○	○	○	○	○	○	○	None	
6-17	strtok	○	○	○	○	○	○	○	○	None	
6-18	memset	○	○	○	○	○	×	○	×	None	
6-19	strerror	○	○	○	○	○	×	○	×	None	
6-20	strlen	○	×	○	×	○	×	○	×	None	
6-21	strcoll	○	×	○	×	○	×	○	×	None	
6-22	strxfrm	○	×	○	×	○	×	○	×	None	
7-01	matherr	○	○	○	○	×	×	×	×	None	math.h
8-01	_ _ assertfail	○	○	○	○	×	×	×	×	Stack Over-flow	assert.h

1-1 is~ Character and String Functions

[Function]

- is~ tests the character type.

[Header file]

- ctype.h for all

[Function prototype]

- int is~ (int c);

Function Name	Argument	Return Value
is~	c - Test character	If the c character is the target character - 1 If the c character is not the target character - 0

[Description]

Function Name	Range
isalpha	Tests whether c is a letter (A to Z, or a to z).
isupper	Tests whether c is an uppercase letter (A to Z).
islower	Tests whether c is a lowercase letter (a to z).
isdigit	Tests whether c is a digit (0 to 9).
isalnum	Tests whether c is an alphanumeric character (0 to 9, A to Z, or a to z).
isxdigit	Tests whether c is a hexadecimal character (0 to 9, A to F, or a to f).
isspace	Tests whether c is a white-space character (space, tab, carriage return, new line, vertical, form feed).
ispunct	Tests whether c is a printing character other than space or an alphanumeric character.
isprint	Tests whether c is a printing character.
isgraph	Tests whether c is a printing character other than a space.
iscntrl	Tests whether c is a control character.
isascii	Tests whether c is an ASCII character.

1-2 toupper tolower

Character and String Functions

[Function]

- Converts the character type.
- toupper converts a lowercase letter to an uppercase letter.
- tolower converts an uppercase letter to a lowercase letter.

[Header file]

- ctype.h

[Function prototype]

- int to~ (int c);

Function Name	Argument	Return Value
toupper tolower	c - Character to be converted	If c can be converted - Character after conversion of c character If c cannot be converted - c

[Description]

toupper

- toupper converts the argument when a lowercase letter into an uppercase letter.

tolower

- tolower converts the argument when an uppercase letter into a lowercase letter.

1-3 toascii — Character and String Functions

[Function]

- toascii converts to ASCII code.

[Header file]

- ctype.h

[Function prototype]

- int toascii (int c);

Function Name	Argument	Return Value
toascii	c - Character to be converted	Value where bits outside the ASCII code range of c are set to 0

[Description]

- Converts to the ASCII code of c. Bits (bits 7 to 15) outside the ASCII code range (bits 0 to 6) are set to 0.

1-4 __toupper/toup __tolower/tolow Character and String Functions

[Function]

- __toupper and toup subtract ‘a’ from c and add ‘A’. a: Lowercase letter
- __tolower and tolow subtract ‘A’ from c and add ‘a’. A: Uppercase letter
  (__toupper and toup, and __tolower and tolow are completely identical.)

[Header file]

- ctype.h

[Function prototype]

- int_to~ (int c);

Function Name	Argument	Return Value
_toupper toup	c - Character to be converted	Value of ‘a’ subtracted from c and added to ‘A’
_tolower tolow		Value of ‘A’ subtracted from c and added to ‘a’

a: Lowercase letter
A: Uppercase letter

[Description]

_toupper

- _toupper resembles toupper, but it does not confirm that the argument is a lowercase letter.

_tolower

- _tolower resembles tolower, but it does not confirm that the argument is an uppercase letter.

2-1 setjmp longjmp

Program Control Functions

[Function]

- setjmp saves the environment during a call.
- longjmp restores the environment saved by setjmp.

[Header file]

- setjmp.h

[Function prototype]

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

Function Name	Argument	Return Value
setjmp	env - Array that saves the environment	If directly called - 0 If returning from the call of the corresponding longjmp - value of val when calling the corresponding longjmp call, but 1 when val is 0
longjmp	env - Array of the environment saved by setjmp val - Return value to setjmp	Since the execution following the setjmp that saved the environment in env is moved to, longjmp is not returned to.

[Description]

setjmp

- If setjmp was directly called, the HL register (for normal model), the DE register (for static model), saddr area used as the register variable, sp, and the return address of the function are saved in env and 0 is returned.

longjmp

- longjmp restores the environment saved in env (HL register (for normal model), the DE register (for static model), saddr used as the register variable, and sp). Program execution continues as if the corresponding setjmp returned val (but is 1 when val is 0).

3-1 va_start va_arg va_end — Special Functions

[Function]

- va_start is for setting a process with a variable number of arguments. (Macro)
- va_arg processes a variable number of arguments. (Macro)
- va_end indicates the end of processing of a variable number of arguments. (Macro)

[Header file]

- stdarg.h

[Function prototype]

- void va_start (va_list ap, parmN);
- type va_arg (va_list ap, type);
- void va_end (va_list ap);

Function Name	Argument	Return Value
va_start	ap - Variable initialized for use in va_arg and va_end parmN - Argument that is one before the variable arguments	None
va_arg	ap - Variable for argument list processing type - Type for pointing to a suitable location in the variable arguments (type is a variable length type. For example, if va_arg(va_list ap, int) is described, type is int. If va_arg(va_list ap, long) is described, type is long.)	If normal - Value of a suitable location in the variable arguments If ap is a null pointer - 0
va_end	ap - Variable for processing a variable number of arguments	None

3-1 va_start va_arg va_end

Special Functions

[Description]

va_start

- In va_start, the ap argument is an object of type va_list (char* type).
- The pointer that points to the argument after parmN is stored in ap.
- parmN is the name of the parameter on the right side in the function definition.
- If parmN is declared in the register storage class, the behavior is undefined.

va_arg

- In va_arg, the ap argument must be identical to ap of type va_list initialized by va_start. (Otherwise, the behavior is undefined.)
- The value of a suitable location in the variable arguments (at the head of the variable argument immediately after va_start and later proceeds to each va_arg) is returned with the type specified by type.
- If ap is a null pointer, 0 having the type specified by type returns.

va_end

- va_end sets ap to the null pointer in order to notify the macro system that the all variable arguments have been processed.

4-1 sprintf — I/O Functions

[Function]

- sprintf writes data into a string in accordance with the format.

[Header file]

- stdio.h

[Function prototype]

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

Function Name	Argument	Return Value
sprintf	s - Pointer to the output string format - Pointer to the string indicating the output conversion specifications ... - Zero or more arguments to be converted	Number of characters written to s (the terminating null character is not counted)

[Description]

- The behavior is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored.

- In accordance with the output conversion specification defined by format, the arguments (0 or more) following format are converted and written to the string pointed to by s.

- The output conversion specification has zero or more directives. Ordinary characters (other than conversion specifications that begin with %) are output unchanged to the string s. The conversion specification fetches successive arguments (0 or more), converts and outputs them to string s.

- Each conversion specification begins with % and is followed in order by the elements shown next. (When the conversion setting is invalid, that character is output. At this time, the flag and the minimum field width are valid.)

- Format
  - Zero or more flags (described later) modify the meaning of the conversion specification.
  - Decimal integer of the option that specifies the minimum field width

If the width after conversion is less than this field width, it is padded on the left. (If the left justification (-) is set, the right side is padded.) There is no padding if the field width begins at 0 and there is right adjustment. Otherwise, the padding is the space character. The width after conversion that exceeds the field width is not discarded.

- Optional precision specification (.integer)
  For d, i, o, u, x, and X conversion, the minimum number of digits is specified. In s conversion, the maximum number of characters is specified. In e, E, and f conversion, the digits that should be output after the decimal-point character is specified. For g and G conversion, the maximum number of significant digits is specified.
  The precision specification is an integer. If the integral part is omitted, the default is 0. The amount of padding produced by the precision setting has precedence over the padding specified by the field width.

- Optional h, l, or L
  h specifies that a following d, i, o, u, x, or X conversion is applied to short int or unsigned short int. In addition, h specifies that a following n conversion is applied to a pointer to short int.
  i specifies that a following d, i, o, u, x, or X conversion is applied to long int or unsigned long int. Also l specifies that a following n conversion is applied to a pointer to long int.
  h, l, and L are ignored for other conversions.

- Characters that specify the conversion (conversion specifications to be described later)
  The field width or precision specification can specify an asterisk (*) instead of an integer string. In this case, the int argument provides an integer value (before the argument to be converted). A negative field width produced by this result is interpreted as a - flag followed by a positive field. Negative precision is ignored.

◊ The flags are:

- – : The conversion result is left justified in the field.

- + : A + or - sign is attached to the result of the signed conversion.

- Space : If there is no sign in the result of the signed conversion, spaces are added at the beginning. If the space and + flags are simultaneously specified, the space flag is ignored.

- # : The result is converted into an “alternate form.”
  For o conversion, the precision is increased so that the first digit becomes zero. For x or X conversion, 0x (or 0X) is prefixed to a non-zero result.
  For e, E, or f conversion, the decimal-point character is always forcibly inserted in the output value. (In the default without #, a decimal-point character appears only if a digit follows.)
  For g or G conversion, the decimal-point character is always forcibly entered in the output value. Trailing zeros are not removed. (In the default without #, the decimal-point character appears only if a digit follows. Trailing zeros are removed.) For other conversions, the # flag is ignored.

4-1 sprintf **I/O Functions**

◊ The conversion specifications are:

- d, i, o, u, x, X : An int argument is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x or X). x conversion uses letters a to f as the hexadecimal characters, and X conversion uses A to F.

The precision specification indicates the minimum number of digits in the result. If the result is too small, leading zeros are added. If the precision is not specified, the default is 1. If precision of 0 is specified and 0 is converted, nothing appears.

- f : A double argument is converted as a signed value in the style [–]dddd.dddd.
  dddd is one or more decimal numbers. The number of digits preceding the decimal point is determined by the absolute value of the number. The number of digits following the decimal point is determined by the required precision. If the precision is omitted, the precision is taken to be 6.

- e : A double argument is converted as a signed value in the style [–]d.dddd e [sign] ddd. d is one decimal digit. dddd is one or more decimal digits. ddd is exactly three decimal digits. sign is + or –.If the precision is omitted, the precision is taken to be 6.

- E : This format is identical to the e format, except E and not e precedes the exponent.

- g : A double argument uses the f or e format depending on whichever produces a more compact result in a conversion based on the specified precision.
  The e format is only used when the exponent of the value is less than –4 or greater than or equal to the number specifying the precision.
  Trailing zeros are removed. The decimal point appears only when followed by one or more digits.

- G : This format is identical to the g format, except E and not e precedes the exponent.

- c : The int argument is converted to unsigned char and the resulting character is written.

- s : The argument is a pointer to a string. Each character from this string is written until the terminating null character (which is not included in the output).
  If the precision is specified, unnecessary characters are not written.
  When the precision is not specified or the precision exceeds the size of the array, the array must include a null character.

- p : The argument is a pointer to void and represented by an unsigned 4-digit hexadecimal number (leading zeros added when less than 4 digits). The large model represents the argument by an unsigned 8-digit hexadecimal number. (The two most-significant digits are padded by 0. Leading zeros are added when there are less than six digits.) The precision specification is ignored.

4-1 sprintf **I/O Functions**

- n : The argument is a pointer to an integer. The number of characters that have been written in the s string are entered in this argument. No conversion is performed.

- %: % is written. No argument is converted.
  (The flag and minimum field width are valid.)

- The behavior for an invalid conversion specifier is undefined.

- If the actual argument is a union or structure, or a pointer to one (except for a character array in a %s conversion or a pointer in a %p conversion), the behavior is undefined.

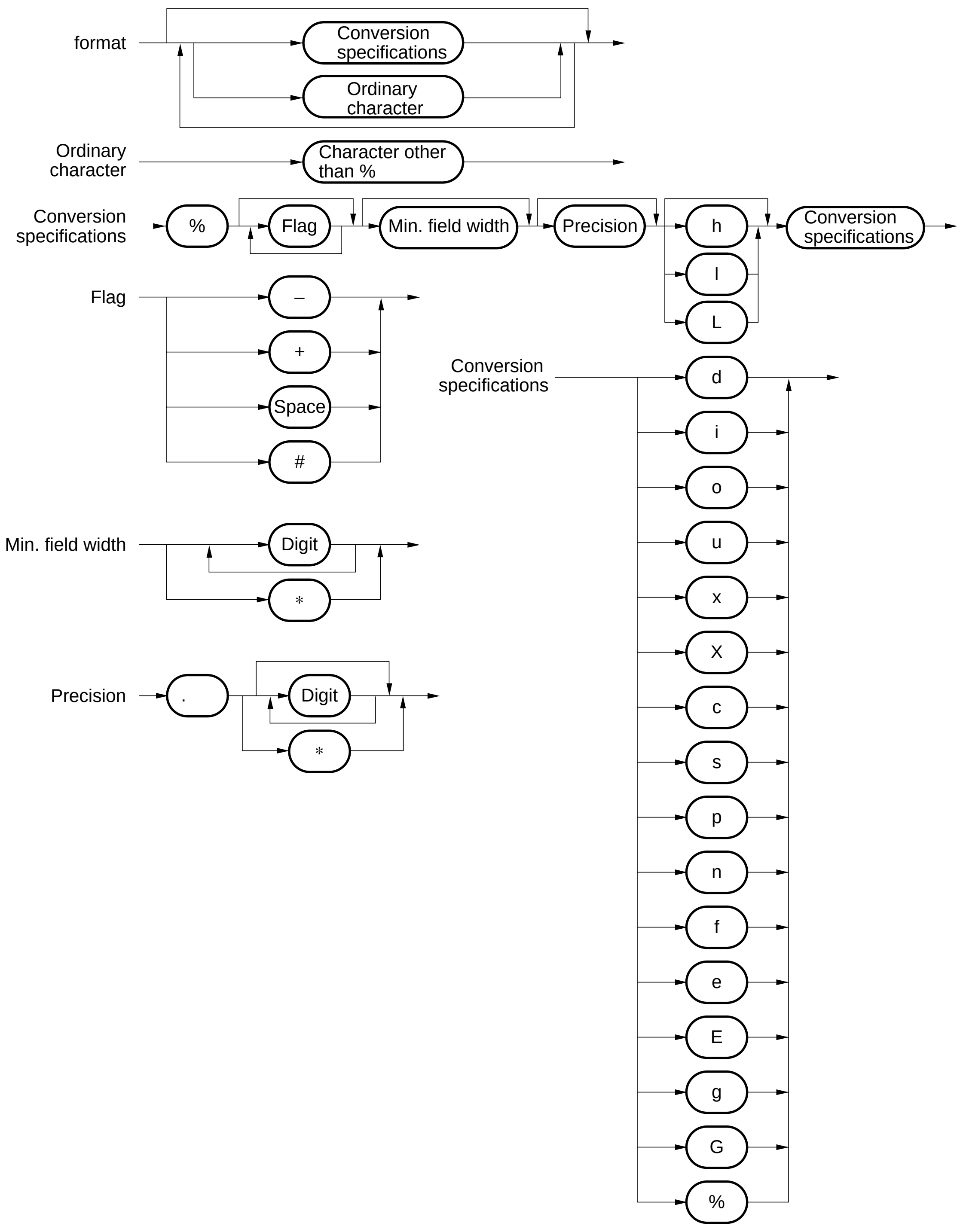
- If there is no field width or it is small, the conversion result is not truncated. That is, when the number of characters in the conversion result is larger than the field width, the field is expanded to a width that holds the conversion result.

- The style of the special output string for %f, %e, %E, %g, or %G conversion are:

Not-a-number	→ “(NaN)”
+∞	→ “(+INF)”
–∞	→ “(–INF)”

A null character is written at the end of the s string (not included in count of the return value).
Figure 10-2 is a syntax chart of format.

Figure 10-2. Syntax Chart for Output format



4-2 sscanf I/O Functions

[Function]

- Data is read in from the input string in accordance with format.

[Header file]

- stdio.h

[Function prototype]

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

Function Name	Argument	Return Value
sscanf	s - Pointer to input string format - Pointer to string indicating the input conversion specification ... - (Zero or more) pointer arguments to objects that input the converted value	If the s string is empty - −1 If the s string is not empty - Number of substituted input items

[Description]

- Input is from the string pointed to by s. The allowed input string is specified by the string pointed to by format. The arguments after format are used as pointers to objects. format specifies how to convert the input string.

- If there are not enough arguments for format, the behavior is undefined. If there excess arguments, the expression is evaluated, but nothing is input.

- format consists of zero or more directives. The directives are:

  (1) One or more white space characters (characters where isspace is true)
  (2) Ordinary characters (not %)
  (3) Conversion specification

- A conversion specification begins with % followed in order by

- An optional assignment-suppressing character * (indicates no assignment to the argument)

- An optional decimal number specifying the maximum field width (no specification when 0)

- An optional h, l, or L (indicates the size of the object on the receiving side)
  If h precedes the d, i, n, o, or x conversion specifier, the argument is not a pointer to int but to short int. If i precedes these conversion specifiers, the argument is a pointer to long int.
  Similarly, if h precedes the u conversion specifier, the argument is a pointer to unsigned short int. If i precedes it, the argument is a pointer to unsigned long int.

4-2 sscanf **I/O Functions**

- If l precedes the e, E, f, g, or G conversion specifier, the argument is a pointer to double. (In the default without l, the argument is a pointer to float.) If L precedes, it is ignored.

  * Conversion specifier : Character indicating the type for the conversion (described later)

sscanf is executed in the order of the directives in format. If a directive fails, sscanf returns.

(1) A directive composed of white-space characters is executed by reading the input until the first non-white-space character (which is not read) or no more characters can be read. If the white-space character directive cannot find a non-white-space character, it fails.

(2) A directive composed of ordinary characters is executed by reading the next characters. If any of those characters differ from the directive characters, the directive fails.

(3) A directive that is a conversion specification defines an input string that matches each conversion specifier (described later). A conversion directive is executed in the following order.

- Input white-space characters (specified by isspace) are skipped, unless the conversion specifier is [, c, or n.

- The input item is read from the s string, unless the conversion specifier is n. The input item is defined as the longest input array in the beginning of the string specified by the conversion specifiers; however, if the maximum field width is specified, the length is truncated. The next character in the input item remains unread. If the length of the input item is zero, the execution of the directive fails.

- Except for the % conversion specifier, the input item (number of input characters for the %n directive) is converted into the type determined by the conversion specifier. If the input item does not match the format, the execution of the directive fails. Unless input is suppressed by *, the conversion result is stored in the object that points to the first argument following the format that has not received a conversion result.

The conversion specifiers are:

- d : Converts to a decimal integer (sign optional). The corresponding argument is a pointer to integer.

- i : Converts to an integer (sign optional). If the number begins with 0x or 0X, it is a hexadecimal integer. If it begins with 0, it is an octal integer. Otherwise, it is a decimal integer. The corresponding argument is a pointer to integer.

- o : Converts to an octal integer (sign optional). The corresponding argument is a pointer to integer.

- u : Converts to an unsigned decimal number. The corresponding argument is a pointer to an unsigned integer.

4-2 sscanf **I/O Functions**

- x : Converts to a hexadecimal integer (sign optional).

- e, E, f, g, G : Floating-point number that consists of an optional sign (+ or −), one or more consecutive decimal digits that include a decimal point, an optional exponent (e or E), followed by an optional signed integer. If the conversion result overflows, it becomes ±∞. If it underflows, it becomes an unnormalized number or ±0. The corresponding argument is a pointer to float.

- s : Input is from a non-white-space characters string. The corresponding argument is a pointer to integer.
  0x or 0X can be prefixed to a hexadecimal integer. The corresponding argument is a pointer to an array that is large enough to hold this string and the terminating null character. The terminating null character is automatically added.

- [: A string is input from a set of expected characters (called the scanset). The corresponding argument is a pointer to an array that is large enough to hold this string and the terminating null character. The terminating null character is automatically added.
  The conversion directive continues from the character after this one until the right bracket (]). The string enclosed by the brackets (called the scanlist) forms the scanset except when the character following the left bracket is a circumflex (^). The circumflex (^) means that the scanset consists of all characters other than those after the circumflex and up to the right bracket in the scanlist. However, if it begins with [ ] or [^], the right bracket is a part of the scanlist, and the next right bracket ends the scanlist. A hyphen (-) anywhere but at the right or left end of the scanlist specifies a range.
  If the ASCII code character to the left of the - is greater than that of the character to the right, the hyphen is simply a character.

- c : A string composed of the number of characters specified by the field width (1 when not specified) is input. The corresponding argument is a pointer to an array that is large enough to hold this string. A terminating null character is not added.

- p : Converts an unsigned hexadecimal number. The corresponding argument is a pointer to a pointer to void.

- n : Nothing is input from the s string. The corresponding argument is a pointer to integer. The number characters already read from the s string by this function are stored in the object pointed to by this pointer.
  The %n directive is not included in the assignment count of the return value.

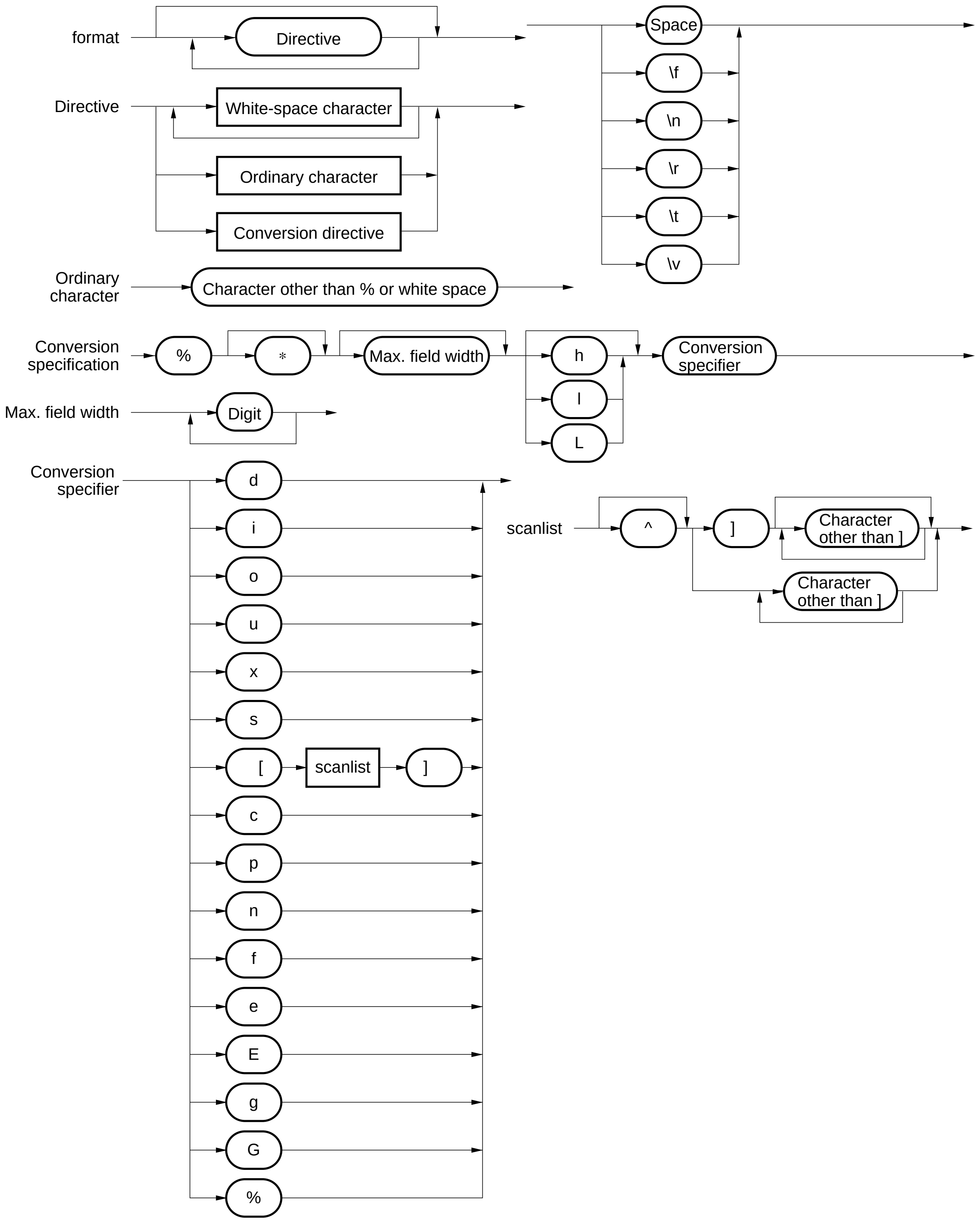
- % : Reads %. No conversions nor assignments are performed.

If a conversion directive is invalid, the directive fails.
If the terminating null character appears in the input string, sscanf returns.
If d, i, o, u, x, or p overflow during integer conversion, the part exceeding the number of bits of the type after conversion are truncated.
The syntax chart for format is shown next.

4-2 sscanf

I/O Functions

Figure 10-3. Syntax Chart for Input format



4-3 printf — I/O Functions

[Function]

- printf outputs data to SFR in accordance with the format.

[Header file]

- stdio.h

[Function prototype]

- int printf (const char *format, ...);

Function Name	Argument	Return Value
printf	format - Pointer to the string indicating by the output conversion specification ... - Zero or more arguments to be converted	Number of characters output to s (terminating null character is not counted)

[Description]

- The arguments (zero or more) following format are conversion in accordance with the output conversion specification described by format and output by using the putchar function.
- The output conversion specification consists of zero or more directives. An ordinary character (not a conversion specification beginning with %) is output unchanged by using putchar. The conversion specification fetches successive arguments (zero or more), and then converts and outputs them by using the putchar function.
- Each conversion specification is identical to the sprintf function.

4-4 scanf — I/O Functions

[Function]

- Reads data in accordance with the format from SFR.

[Header file]

- stdio.h

[Function prototype]

- int scanf (const char *format, ...);

Function Name	Argument	Return Value
scanf	format - Pointer to string indicating the input conversion specification ... - (Zero or more) pointer arguments to objects that input the converted values	When the s string is not empty - Number of assigned input items

[Description]

- The getchar function is used to perform input. The allowed input string is specified by the string pointed to by format. The arguments after format are used as pointers to the objects. format specifies how to convert the input string.
- When there are insufficient arguments for format, the behavior is undefined. If there are too many arguments, the expression is evaluated, but is not input.
- The format consists of zero or more directives. The directives are

  (1) One or more white-space characters (character where isspace is true)
  (2) Ordinary characters (except %)
  (3) Conversion directive

- If conversion is terminated by an input character that is invalid for the directive, the invalid input character is discarded. The conversion directive is identical to the sscanf function.

4-5 vprintf I/O Functions

[Function]

- vprintf outputs data to SFR in accordance with the format.

[Header file]

- stdio.h

[Function prototype]

- int vprintf (const char *format, va_list p);

Function Name	Argument	Return Value
printf	format - Pointer to string indicating the output conversion specification p - Pointer to argument sequence	Number of characters output (terminating null character is not counted)

[Description]

- The argument pointed to by the pointer in the argument sequence is converted in accordance with the output conversion specification designated by format and output by using the putchar function.
- Each conversion specification is identical to the sprintf function.

4-6 vsprintf — I/O Functions

[Function]

- vsprintf writes data to a string in accordance with the format.

[Header file]

- stdio.h

[Function prototype]

- int vsprintf (char *s, const char *format, va_list p);

Function Name	Argument	Return Value
printf	s - Pointer to a string to be output format - Pointer to the string designating the output conversion specification p - Pointer to argument sequence	Number of characters output to s (terminating null character not counted)

[Description]

- The arguments pointed to by the pointers in the argument sequence are output to the string pointed to by s in accordance with the output conversion specification designated by format.
- The output conversion specification is identical to the sprintf conversion.

4-7 getchar I/O Functions

Function]

- One character is read from SFR.

[Header file]

- stdio.h

[Function prototype]

- int getchar (void);

Function Name	Argument	Return Value
getchar		One character read from SFR

[Description]

- The value read from the SFR symbol P0 (port 0) is returned.
- No error checking is performed for the read.
- If the SFR to be read is changed, the source must be modified and registered again in the library, or the user must create a new getchar function.

4-8 gets **I/O Functions**

[Function]

- Reads a string.

[Header file]

- stdio.h

[Function prototype]

- char *gets (char *s);

Function Name	Argument	Return Value
gets	s - Pointer to input string	If normal - s If end-of-file is detected before any character is read - Null pointer

[Description]

- The string is read by the getchar function and stored in the array pointed to by s.
- When the end-of-file is detected (when getchar returns -1), or when a new line character is read, reading the string ends. The new line character that was read is discarded and a null character is written after the last character stored in the array.
- When correct, s returns.
- If the end of the file is detected before reading any characters in the array, the contents of the array are not changed, and the null pointer is returned.

4-9 putchar — I/O Functions

[Function]

- One character is output to SFR.

[Header file]

- stdio.h

[Function prototype]

- int putchar (int c);

Function Name	Argument	Return Value
putchar	c - Output character	Output character

[Description]

- The character specified by c in the SFR symbol P0 (port 0) is converted to unsigned char type and written.
- No error checking is performed for the read.
- If the SFR to be written is changed, the source must be modified and registered again in the library, or the user must create a new putchar function.

4-10 puts **I/O Functions**

[Function]

- A string is output.

[Header file]

- stdio.h

[Function prototype]

- int puts (const char *s);

Function Name	Argument	Return Value
puts	s - Pointer to the output string	If normal - 0 If putchar returned −1 - −1

[Description]

- The putchar function is used to write the string pointed to by s. The new line character is added to the end of the output.
- The terminating null character of the string is not written.
- If normal, 0 is returned. If putchar returned −1, −1 is returned.

5-1 atoi
atol — Utility Functions

[Function]

- atoi converts a decimal constant string to int.
- atol converts a decimal constant string to long.

[Header file]

- stdlib.h

[Function prototype]

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

Function Name	Argument	Return Value
atoi	nptr - Converted string	If normal - Converted value If a positive overflow occurs - INT_MAX(32767) If a negative overflow occurs - INT_MIN(-32768) If an invalid string - 0
atol		If normal - Converted value If a positive overflow occurs - LONG_MAX(2147483647) If a negative overflow occurs - LONG_MIN(-2147483648) If an invalid string - 0

[Description]

atoi

- The first part of the string pointed to by nptr is converted to int.
- An array of zero or more white-space characters (character where isspace is true) at the beginning is skipped. The array of decimal digits following an optional sign after the next character (until a character other than a decimal digit or a terminating null character appears) is converted to integer. If there are no decimal digits, 0 is returned. If an overflow occurs, INT_MAX(32767) is returned when positive, and INT_MIN(–32768) is returned when negative.

atol

- The first part of the string pointed to by nptr is converted to long.
- An array of zero or more white-space characters (character where isspace is true) at the beginning is skipped. The array of decimal digits following an optional sign after the next character (until a character other than a decimal digit or a terminating null character appears) is converted to integer. If there are no decimal digits, 0 is returned. If an overflow occurs, LONG_MAX(2147483647) is returned when positive, and LONG_MIN(–2147483648) is returned when negative.

5-2 strtol strtoul — Utility Functions

[Function]

- strtol converts a string to long.
- strtoul converts a string to unsigned long.

[Header file]

- stdlib.h

[Function prototype]

- long int strtol (const char *nptr, char **endptr, int base);
- unsigned long int strtoul (const char *nptr, char **endptr, int base);

Function Name	Argument	Return Value
strtol	nptr - Converted string endptr - Pointer storing the pointer to the unrecognizable part base - Specified base	If normal - Converted value If a positive overflow occurs - LONG_MAX(2147483647) If a negative overflow occurs - LONG_MIN(-2147483648) If not converted - 0
strtoul		If normal - Converted value If an overflow occurs - ULONG_MAX(4294967295U) If not converted - 0

[Description]

strtol

- The string pointed to by nptr is decomposed into these three parts.

  (1) White-space string (specified by isspace) even if empty
  (2) Integer expression in the base set by the value in base
  (3) Sequence of one or more unrecognized characters (including terminating null character)

  The string in (2) is converted to integer and the result is returned.

- If base is 0, a C numerical expression is interpreted (0x- or 0X- (hexadecimal number), 0- (octal number), digit other than 0 (decimal number), a sign is optional).
- If base is 2 to 36, it becomes the base (can begin with a sign). a (or A) to z (or Z) represents 10 to 35. If the base is 16, 0x or 0X may follow the sign (if present).
- The pointer to the string in (3) (if endptr is not a null pointer) is stored in the object pointed to by endptr.
- If there is positive overflow, LONG_MAX(2147483647) is returned. If there is negative overflow, LONG_MIN(–2147483648) is returned. ERANGE(2) is set in errno.
- If the string in (2) is empty or does not have the expected format, without performing any conversion, the pointer to the string is stored in the object pointed to by endptr (if endptr is not a null pointer), and 0 is returned. This is similar to the cases where the base is not 0 or 2 to 36.

5-2 strtol
strtoul **Utility Functions**

strtoul

- The string pointed to by nptr is decomposed into three parts.

  (1) White-space character string (specified by isspace) even if empty
  (2) Integer expression in the base set by the value in base
  (3) Sequence of one or more unrecognized characters (including terminating null character)

  The string in (2) is converted to unsigned integer and the result is returned.

- If base is 0, a C numerical expression is interpreted (0x- or 0X- (hexadecimal number), 0- (octal number), digit other than 0 (decimal number)).
- If base is 2 to 36, it becomes the base. a (or A) to z (or Z) represents by 10 to 35. If the base is 16, 0x or 0X may be added.
- The pointer to the string in (3) (if endptr is not a null pointer) is stored in the object pointed to by endptr.
- If there is overflow, ULONG_MAX(4294967295U) is returned. ERANGE(2) is set in errno.
- If the string in (2) is empty or does not have the expected format, without performing any conversion, the pointer to the string is stored in the object pointed to by endptr (if endptr is not a null pointer), and 0 is returned. This is similar to the cases where the base is not 0 or 2 to 36.

5-3 calloc — Utility Functions

[Function]

- calloc allocates an array space and initializes it to 0.

[Header file]

- stdlib.h

[Function prototype]

- void *calloc (size_t nmemb, size_t size);

Function Name	Argument	Return Value
calloc	nmemb - Number of objects in the array size - Size of the array	If allocated - Pointer to the beginning of the allocated space If not allocated - Null pointer

[Description]

- Space for nmemb objects is allocated in the array of size bytes and initialized to 0.
- The pointer to the beginning of the allocated space is returned.
- If space is not allocated, the null pointer is returned.
- Allocation begins at the break value. The address following the allocated space becomes the new break value. The break value is specified by brk. For information on brk, see "**5-11 brk.**"

5-4 free **Utility Functions**

[Function]

- Frees an allocated block.

[Header file]

- stdlib.h

[Function prototype]

- void free (void *ptr);

Function Name	Argument	Return Value
free	ptr - Pointer to the beginning of the block to be freed	None

[Description]

- Space allocated (up to the break value) from the space pointed to by ptr is freed. (Calling malloc, calloc, or realloc after free allocates space from ptr.)
- If ptr does not point to allocated space, nothing happens. (Freeing is performed by setting ptr to a new break value.)

5-5 malloc — Utility Functions

[Function]

- malloc allocates a block.

[Header file]

- stdlib.h

[Function prototype]

- void *malloc (size_t size);

Function Name	Argument	Return Value
malloc	size - Size of the allocated block	If allocated - Pointer to the beginning of the allocated space If not allocated - Null pointer

[Description]

- Space having size bytes is allocated. The pointer to the beginning of the allocated space is returned.
- If nothing is allocated, the null pointer is returned.
- Allocation begins at the break value. The address following the allocated space becomes the new break value. The break value is specified by brk. For information on brk, see “**5-11 brk**.”

5-6 realloc

Utility Functions

[Function]

- realloc reallocates a block.

[Header file]

- stdlib.h

[Function prototype]

- void *realloc (void *ptr, size_t size);

Function Name	Argument	Return Value
realloc	ptr - Pointer to the beginning of the block to be reallocated size - Size of the block being reallocated	If reallocated - Pointer to the beginning of the reallocated space If allocated with ptr that is a null pointer - Pointer to the beginning of the allocated space If reallocation or allocation is not possible - Null pointer

[Description]

- This size of the space allocated from the space pointed to by ptr (until the break value) changes to size. The contents are not changed in the smaller of the space to be allocated and the allocated space to be reallocated. If the size is increased, the additional part is allocated. If decreased, the reduced part is freed.
- If ptr is a null pointer, a new space specified by size is allocated (same as malloc).
- If ptr does not point to the allocated space, or nothing is allocated, the null pointer is returned without doing anything.
- Reallocation makes the address of size bytes added to ptr the new break value.

5-7 abort — Utility Functions

[Function]

- abort aborts the program.

[Header file]

- stdlib.h

[Function prototype]

- void abort (void);

Function Name	Argument	Return Value
abort	None	Does not return.

[Description]

- Does not loop and return.
- The user creates abort processing.

5-8 atexit exit

Utility Functions

[Function]

- atexit registers the function to be called upon normally exiting a program.
- exit terminates a program

[Header file]

- stdlib.h

[Function prototype]

- int atexit (void (*func)(void));
- void exit (int status);

Function Name	Argument	Return Value
atexit	func - Pointer to the function to be registered	If function registration was successful - 0 If the function cannot be registered - 1
exit	status - Value indicating the exit status	Does not return.

[Description]

atexit

- atexit registers that the function pointed to by func is called without arguments when the program exits normally.
- Up to 32 functions can be registered. If registration was successful, 0 is returned. If 32 functions are registered and no more can be registered, 1 is returned without any registration.

exit

- exit terminates the program normally.
- The functions registered initially by atexit are called in reverse order of registration.
- Does not loop and return.
- The user writes the exit processing.

5-9 abs labs

Utility Functions

[Function]

- abs determines the absolute value of a value of type int.
- labs determines the absolute value of a value of type long.

[Header file]

- stdlib.h

[Function prototype]

- int abs (int j);
- long int labs (long int j);

Function Name	Argument	Return Value
abs	j - Value whose absolute value is taken	If –32767 ≤ j ≤ 32767 - Absolute value of j If j is –32768 - –32768 (0x8000)
labs		If –2147483647 ≤ j ≤ 2147483647 - Absolute value of j If j is –2147483648 - –2147483648 (0x80000000)

[Description]

abs

- abs determines the absolute value of the value of j (int type).
- If j is -32768, -32768 is returned.

labs

- labs determines the absolute value of the value of j (long type).
- If j is -2147483648 -2147483648 is returned.

5-10 div ldiv

Utility Functions

[Function]

- div performs division on type int to find the quotient and remainder.
- ldiv performs division on type long to find the quotient and remainder.

[Header file]

- stdlib.h

[Function prototype]

- div_t div (int numer, int denom);
- ldiv_t ldiv (long int numer, long int denom);

Function Name	Argument	Return Value
div	numer - Numerator denom - Denominator	The quotient is returned in member quot and the remainder in rem in type div_t.
ldiv		The quotient is returned in member quot and the remainder in rem in type ldiv_t.

[Description]

div

- div determines the quotient and remainder of numer divided by denom.
- The absolute value of the quotient is the maximum integer less than the absolute value of the numerator divided by the absolute value of the denominator. The sign is the same as the number (positive when the signs of both numer and denom are positive, negative when they differ).
- The remainder is the value of the numer - denom * quotient.
- If denom is 0, the quotient is 0 and the remainder is numer.
- If numer is –32768 and denom is –1, the quotient is 32768 and the remainder is 0.

ldiv

- ldiv determines the quotient and remainder of numer divided by denom.
- The absolute value of the quotient is the maximum integer (type long int) less than the absolute value of the numer divided by the absolute value of the denom. The sign is the same as the number (positive when the signs of both numer and denom are positive, negative when they differ).
- The remainder is the value of the numer - denom * quotient.
- If denom is 0, the quotient is 0 and the remainder is numer.
- If numer is –2147483648 and denom is –1, the quotient is 2147483648 and the remainder is 0.

5-11 brk
sbrk **Utility Functions**

[Function]

- brk sets the break value.
- sbrk increments or decrements the break value.

[Header file]

- stdlib.h

[Function prototype]

- int brk (char *endds);
- char *sbrk (int incr);

Function Name	Argument	Return Value
brk	endds - Break value to be set	If normal - 0 If break value cannot be changed - −1
sbrk	incr - Amount to increase or decrease the break value	If normal - Former break value If the break value cannot be increased or decreased - −1

[Description]

brk

- brk sets the value given by endds to the break value.
- If endds is outside the scope, the break value is not changed and ENOMEM(3) is set in errno.

sbrk

- sbrk increases or decreases the break value by incr bytes (based on the sign of incr).
- If the break value after the increase or decrease is outside the scope, the break value does not change and ENOMEM(3) is set in errno and −1 is returned.

5-12 itoa ltoa ultoa — Utility Functions

[Function]

- itoa converts int to a string.
- ltoa converts long to a string.
- ultoa converts unsigned long to a string.

[Header file]

- 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 Name	Argument	Return Value
itoa	value - Value being converted	If normal - Pointer to the converted string
ltoa	string - Pointer to the conversion result	
ultoa	radix - Specified base	Otherwise - Null pointer

[Description]

itoa, ltoa, ultoa

- The specified number value is converted into a string terminated by a null character. The result is stored in the space pointed to by string. Conversion is performed with the specified base radix. The pointer to the converted string is returned.
- radix must be in the range from 2 to 36. Otherwise, conversion is not performed and the null pointer is returned.

5-13 rand srand — Utility Functions

[Function]

- rand generates a pseudo-random number.
- srand seeds the generation status for the pseudo-random numbers.

[Header file]

- stdlib.h

[Function prototype]

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

Function Name	Argument	Return Value
rand	None	Pseudo-random number in the range from 0 to RAND_MAX
srand	seed - Initial value for the generation status for pseudo-random numbers	None

[Description]

rand

- rand generates a pseudo-random number in the range from 0 to RAND_MAX.

srand

- srand seeds the generation status for pseudo-random numbers. The seed is used as the value that becomes the base for the sequence of pseudo-random numbers that is returned when the rand function is called. For the same seed values, the sequence of pseudo-random numbers does not change even if srand is called again.
- If rand is called without calling srand, it is identical to calling rand after calling srand with seed=1.

5-14 bsearch Utility Functions

[Function]

- bsearch performs a binary search.

[Header file]

- 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 Name	Argument	Return Value
bsearch	key - Pointer to the search value base - Pointer to the search array nmemb - Number of array objects size - Size of one object in the array compare - Function that compares the key to the array objects and returns their relationship	If the array object matches - Pointer to the first array object that matched If no array object matches - Null pointer

[Description]

- A binary search is performed on the array pointed to by the base pointer for the object pointed to by key. The array pointed to by the base pointer is the array whose nmemb objects having size for the size and sorted in ascending order.
- The compare function compares the object pointed to by key and an array object. The relationship is returned by one of the following values. The first argument of compare is key and the second is the array object.

Less than 0	- Object pointed to by key is smaller
0	- Both elements are equal
Greater than 0	- Object pointed to by key is larger

5-15 qsort — Utility Functions

[Function]

- qsort performs a quick sort.

[Header file]

- stdlib.h.

[Function prototype]

- void qsort (void *base, size_t nmemb, size_t size, int (*compare)(const void*, const void*));

Function Name	Argument	Return Value
qsort	base - Pointer to the array to be sorted nmemb - Number of array objects size - Size of one object in the array compare - Function that compares two array objects and returns their relationship	None

[Description]

- A quick sorts an array pointed to by the base pointer in ascending order. The array pointed to by the base pointer is an array of nmemb objects having the size of size.
- The compare function compares two array elements (array elements 1 and 2), and their relationship is returned by one of the following values.
- The first augument of the compare function is array element 1 and the second augument, array element 2.

Less than 0	-	The first array element is smaller.
0	-	The elements are equal.
Greater than 0	-	The first array element is larger.

- If the array elements are equal, the one closest to the beginning of the array is first.

5-16 strbrk **Utility Functions**

[Function]

- Sets the break value.

[Header file]

- stdlib.h

[Function prototype]

- int strbrk (char *endds);

Function Name	Argument	Return Value
strbrk	endds - Break value to be set	If normal - 0 If the break value cannot be changed - −1

[Description]

- The value given by endds is set to the break value (address following the last address in the allocated space)
- When endds is outside the allowable range, the break value is not changed, ENOMEM(3) is set in errno, and −1 is returned.

5-17 strsbrk Utility Functions

[Function]

- The break value is increased or decreased.

[Header file]

- stdlib.h

[Function prototype]

- char *strsbrk (int incr);

Function Name	Argument	Return Value
strsbrk	incr - Increase or decrease in the break value	If normal - Prior break value If the break value cannot be increased or decreased - −1

[Description]

- The break value is increased or decreased by incr bytes (depending on the sign of incr).
- If the break value after the increase or decrease is outside the scope, ENOMEM(3) is set in errno without changing the break value, and −1 is returned.

5-18 stritoa strltoa strultoa — Utility Functions

[Function]

- stritoa converts int to a string.
- strltoa converts long to a string.
- strultoa converts unsigned long to a string.

[Header file]

- 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 Name	Argument	Return Value
stritoa strltoa strultoa	value - String to be converted string - Pointer to the conversion result radix - Specified radix	If normal - Pointer to the converted string Otherwise - Null pointer

[Description]

stritoa, strltoa, strultoa

- The specified number value is converted into a null-terminated string. The result is stored in the area pointed to by string. The conversion is performed with the specified base radix and returns the pointer to the converted string.
- radix must be in the range from 2 to 36. Otherwise, the null pointer is returned without any conversion.

6-1 memcpy memmove — String/Memory Functions

[Function]

- memcpy copies the specified number of characters in one buffer to another.
- memmove copies the specified number of characters in one buffer to another (even if the buffers overlap, the operation is correct).

[Header file]

- string.h

[Function prototype]

- void *memcpy (void *s1, const void *s2, size_t n);
- void *memmove (void * s1, const void *s2, size_tn);

Function Name	Argument	Return Value
memcpy memmove	s1 - Pointer to the first object in the copy destination s2 - Pointer to the first object in the copy source n - Specified number of characters	s1 value

[Description]

memcpy

- memcpy copies n characters in the object pointed to by s2 to the object pointed to by s1.
- If s2 < s1 < s2+n, the behavior is undefined (since copying is in order from the beginning).

memmove

- memmove copies n characters in the object pointed to by s2 to the object pointed to by s1.
- If the objects pointed to by s1 and s2 overlap, the behavior is correct.

6-2 strcpy strncpy — String/Memory Functions

[Function]

- strcpy copies a string.
- strncpy copies the specified number of characters from the beginning of the string.

[Header file]

- string.h

[Function prototype]

- char *strcpy (char *s1, const char *s2);
- char *stmcpy (char *s1, const char *s2, size_t n);

Function Name	Argument	Return Value
strcpy strncpy	s1 - Pointer to the copy destination string s2 - Pointer to the copy source string n - Number of characters to be copied	s1 value

[Description]

strcpy

- strcpy copies the string pointed to by s2 (including the terminating null character) to the string pointed to by s1.
- If s2 < s1 ≤ s2+ (length of the string to be copied), the behavior is undefined (because copying is in order from the beginning).

strncpy

- strncpy copies n characters in the string pointed to s2 to the string pointed to by s1.
- If s2 < s1 ≤ (minimum of s2 + length of the string to be copied, or s2 + n – 1), the behavior is undefined (because copying is in order from the beginning).
- If the string pointed to by s2 is less than n characters, copying stops at the terminating null character. If there are n or more characters, the first n characters are copied and the terminating null character is not copied.

6-3 strcat strncat

Character String/Memory Functions

[Function]

- strcat appends a string to another string.
- strncat appends the specified number of characters of a string to another string.

[Header file]

- string.h

[Function prototype]

- char *strcat (char *s1, const char *s2);
- char *strncat (char *s1, const char *s2, size_t n);

Function Name	Argument	Return Value
strcat	s1 - Pointer to the destination array	s1 value
	s2 - Pointer to the source array	
strncat	n - Number of appended characters	

[Description]

strcat

- strcat copies the string pointed to by s2 (including the terminating null character) to the end of the string pointed to by s1. The first character in s2 overwrites the null character in s1.
- If overlapping objects are copied, the behavior is undefined.

strncat

- strncat appends n characters in the string pointed to by s2 (not including the terminating null character) to the end of the string pointed to by s1. The first character in s2 overwrites the terminating character in s1.
- If the string pointed to by s2 is less than n characters, appending ends at the terminating null character. If n or more characters, the first n characters are appended.
- A terminating null character is always appended.
- If overlapping objects are copied, the behavior is undefined.

6-4 memcmp Character String/Memory Functions

[Function]

- memcmp compares the specified characters in two buffers.

[Header file]

- string.h

[Function prototype]

- int memcmp (const void *s1, const void *s2, size_tn);

Function Name	Argument	Return Value
memcmp	s1 - Pointer to a comparison object s2 - Pointer to a comparison object n - Number of characters to be compared	If s1 and s2 have n equal characters - 0 If some of the n characters in s1 and s2 differ - - Difference between the first differing characters converted to int (s1 character – s2 character)

[Description]

- n characters in the object pointed to by s1 and the object pointed to by s2 are compared.
- If the n characters in s1 and s2 are equal, 0 is returned.
- If characters differ in the n characters in s1 and s2, the difference between the first differing characters converted to int (s1 character – s2 character) is returned.

6-5 strcmp strncmp

Character String/Memory Functions

[Function]

- strcmp compares two strings.
- strncmp compares the specified portions of two strings.

[Header file]

- string.h

[Function prototype]

- int strcmp (const char *s1, const char *s2);
- int stmcmp (const char *s1, const char *s2, size_t n);

Function Name	Argument	Return Value
strcmp	s1 - Pointer to a comparison string s2 - Pointer to a comparison string	If strings s1 and s2 are equal - 0 If strings s1 and s2 differ - Difference between the first differing characters converted to int (s1 character – s2 character)
stmcmp	s1 - Pointer to a comparison string s2 - Pointer to a comparison string n - Number of characters to be compared	If n characters in strings s1 and s2 are equal - 0 If there are differences in the n characters in strings s1 and s2 - Difference between the first differing characters converted to int (s1 character – s2 character)

[Description]

strcmp

- strcmp compares a string pointed to by s1 to a string pointed to by s2.
- If strings s1 and s2 are equal, 0 is returned. If strings s1 and s2 differ, the difference between the first differing characters (character in s1 – character in s2), that has been converted to int, is returned.

strncmp

- strncmp compares n characters in strings s1 and s2.
- If n characters in strings s1 and s2 are equal, 0 is returned. If there are differences in the n characters in strings s1 and s2, the difference between the first differing characters (character in s1 – character in s2), that has been converted to int, is returned.

6-7 memchr **Character String/Memory Functions**

[Function]

- memchr searches for the specified character in a buffer having the specified number of characters.

[Header file]

- string.h

[Function prototype]

- void *memchr (const void *s, int c, size_t n);

Function Name	Argument	Return Value
memchr	s - Pointer to the object to be searched c - Specified character n - Number of characters in the object to be searched	If the character c is present - Pointer to the first occurrence of the character c If the character c is not present - Null pointer

[Description]

- Returns the pointer to the position of the first occurrence of c (converted to unsigned char) in first n characters of the object pointed to by s.
- If not found, the null pointer is returned.

6-8 strchr strrchr

Character String/Memory Functions

[Function]

- strchr searches for the specified character in the string and returns the location of the first occurrence.
- strrchr searches for the specified character in the string and returns the location of the last occurrence.

[Header file]

- string.h

[Function prototype]

- char *strchr (const char *s, int c);
- char *strrchr (const char *s, int c);

Function Name	Argument	Return Value
strchr strrchr	s - Pointer to the string to be searched c - Specified character	If the character c is in the s string - Pointer to the character c found first or last in the s string If the character c is not in string s - Null pointer

[Description]

strchr

- strchr determines the location of the first occurrence of c (converted to char type) in the string pointed to by s and returns its pointer.
- The terminating null character is regarded as a part of the string.
- If the character c is not in string s, the null pointer is returned.

strrchr

- strrchr determines the location of the last occurrence of c (converted to char type) in the string pointed to by s and returns its pointer.
- The terminating null character is regarded as a part of the string.
- If the character c is not in the s string, the null pointer is returned.

6-9 strspn strcspn

Character String/Memory Functions

[Function]

- strspn determines the length from the beginning of a part that consists only of the characters in the specified string in the string to be searched.
- strcspn determines the length from the beginning of a part that consists of the characters not in the specified string in the string to be searched.

[Header file]

- string.h

[Function prototype]

- size_t strspn (const char *s1, const char *s2);
- size_t strcspn (const char *s1, const char *s2);

Function Name	Argument	Return Value
strspn	s1 - Pointer to the string to be searched	Length of the part consisting only of the characters specified by s2 in string s1
strcspn	s2 - Pointer to the string indicating the specified string	Length of the part consisting of the characters not specified by s2 in string s1

[Description]

strspn

- strspn returns the length of the part consisting only of the characters included in the string pointed to by s2 in the string pointed to by s1.
- The terminating null character in s2 is not considered as a part of s2.

strcspn

- strcspn returns the length of the part not consisting of the characters included in the string pointed to by s2 in the string pointed to by s1.
- The terminating null character in s2 is not considered as a part of s2.

6-10 strpbrk — Character String/Memory Functions

[Function]

- strpbrk finds the location of the first occurrence of any character in the specified string in the string being searched.

[Header file]

- string.h

[Function prototype]

- char *strpbrk (const char *s1, const char *s2);

Function Name	Argument	Return Value
strpbrk	s1 - Pointer to the string to be searched s2 - Pointer to the string indicating the specified character	If any character in string s2 is in string s1 - Pointer to the first occurrence in string s1 of any character in string s2 If the character in string s2 is not in string s1 - Null pointer

[Description]

- Determines the location of the first occurrence in the string pointed to by s1 of any character in the string pointed to by s2 and returns its pointer.
- If no character in string s2 is in string s1, the null pointer is returned.

6-11 strstr Character String/Memory Functions

[Function]

- strstr determines the location of the first occurrence of the specified string in the string to be searched.

[Header file]

- string.h

[Function prototype]

- char *strstr (const char *s1, const char *s2);

Function Name	Argument	Return Value
strstr	s1 - Pointer to the string to be searched s2 - Pointer to the specified string	If s2 string is in string s1 - Pointer to the first location where s2 string first occurs in string s1 If s2 string is not in string s1 - Null pointer If s2 is an empty string - Value of s1

[Description]

- Returns the pointer to the first location where the string pointed to by s2 (except the terminating null character) exactly matches the string pointed to by s1.
- If the s2 string is not in the s1 string, the null pointer is returned.
- If s2 points to an empty string, the value of s1 is returned.

6-12 strtok Character String/Memory Functions

[Function]

- A string is broken into strings composed of characters which are not delimiters.

[Header file]

- string.h

[Function prototype]

- char *strtok (char *s1, const char *s2);

Function Name	Argument	Return Value
strtok	s1 - Pointer to the string to be separated or the null pointer s2 - Pointer to the string indicating the delimiting characters of the tokens	If a token - Pointer to the first character in the token If not a token - Null pointer

[Description]

- A token is a string composed of characters other than the delimiting characters in the specified string.
- If s1 is a null pointer, the string pointed to by the pointer saved in the previous strtok call becomes the string to be separated. However, if the saved pointer is the null pointer, nothing is done and the null pointer is returned.
- If s1 is not a null pointer, the string pointed to by s1 is the string to be separated.
- A character not included in the string pointed to by s2 is searched for in the string to be separated. If not found, the saved pointer becomes the null pointer, and the null pointer is returned. If found, that character becomes the first character in the token.
- If the first character of the token is found, the characters in the s2 string are searched for after the first character in the token. If none is found, the saved pointer becomes the null pointer. If any is found, the null pointer is written at that character position. The pointer to the next character becomes the saved pointer.
- The pointer to the first character of the token is returned.

6-13 memset — Character String/Memory Functions

[Function]

- memset initializes the specified number of characters in the buffer to the specified characters.

[Header file]

- string.h

[Function prototype]

- void *memset (void *s, int c, size_t n);

Function Name	Argument	Return Value
memset	s - Pointer to the object to be initialized c - Specified character n - Number of specified characters	Value of s

[Description]

- The value of c (converted to unsigned char) is copied into the first n characters in the object pointed to by s.

6-14 strerror — Character String/Memory Functions

[Function]

- strerror returns a pointer to the space that saves the error message corresponding to the specified error number.

[Header file]

- string.h

[Function prototype]

- char *strerror (int errnum);

Function Name	Argument	Return Value
strerror	errnum - Error number	If there is an error corresponding to the error number - Pointer to the error message string If there is no error corresponding to the error number - Null pointer

[Description]

- A pointer to the next string corresponding to the errnum value is returned.

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

Otherwise, the null pointer is returned.

6-15 strlen Character String/Memory Functions

[Function]

- Determines the length of the string.

[Header file]

- string.h

[Function prototype]

- size_t strlen (const char *s);

Function Name	Argument	Return Value
strlen	s - Pointer to the string	Length of the s string

[Description]

- The number of characters in the string pointed to by s is returned. The number of characters is the number from the character beginning the string to the character preceding the terminating null character.

6-16 strcoll Character String/Memory Functions

[Function]

- A string is copied.

[Header file]

- string.h

[Function prototype]

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

Function Name	Argument	Return Value
strcoll	s1 - Pointer to a comparison string s2 - Pointer to a comparison string	If strings s1 and s2 are equal - 0 If strings s1 and s2 differ - Difference between the first differing characters converted to int (s1 character − s2 character)

[Description]

- The operation is identical to strcmp.

6-17 strxfrm Character String/Memory Functions

[Function]

- The string is transformed based on unique area information.

[Header file]

- string.h

[Function prototype]

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

Function Name	Argument	Return Value
strxfrm	s1 - Pointer to a comparison string s2 - Pointer to a comparison string n - Maximum number of characters entered in s1	Returns the length of the transformed string (not including the terminating null character) If the return value is N or greater, the contents of the array pointed to by s1 is undefined.

[Description]

- This operation is identical to the following.

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

7-1 matherr — Mathematical Functions

[Function]

- Processes the exceptions of the library that handles floating-point numbers.

[Header file]

- math.h

[Function prototype]

- void matherr (struct exception *x);

Function Name	Argument	Return Value

matherr	struct exception {
	int type;
	char *name;
	}
	
	type - Value indicating the exception type
	name - Function name
		None

[Description]

- This function is called when an exception occurs in the standard library and run-time library that handle floating-point numbers.
- If called from the standard library, EDOM or ERANGE is set in errno. This table shows the relationship between the exception type and errno.

type	Exception Type	Value Set in errno
1	Underflow	ERANGE
2	Loss of significance	ERANGE
3	Overflow	ERANGE
4	Division by zero	EDOM
5	Cannot compute	EDOM

Modifying or creating matherr enables a special error processing.

8-1 _ _assertfail — Diagnostic Function

[Function]

- The _ _assertfail function receives information from the assert macro, calls the printf function, outputs information, and calls up the abort funtion.

[Header file]

- assert.h

[Function prototype]

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

Function Name	Argument	Return Value
_ _assertfail	_ _ msg - Pointer to the string designating the output conversion specification to be passed to the printf function _ _cond - Argument of the assert macro _ _file - Source file name _ _line - source line number	Undefined

[Description]

- The assert macro adds a diagnostic function to the program. When the assert macro is executed, if p is false (in other words, equal to 0), the assert macro passes information concerning a specific call that has produced a false value to _ _assertfail (this information includes the text of the argument, the source file name and the source line number. The last two items are the values of macro_ _FILE and _ _ LINE _ _, respectively).

CHAPTER 11 EXTENDED FUNCTIONS

This chapter describes the extended functions unique to this C compiler that do not conform to the American National Standards Institute (ANSI) standard.

The extended functions of this C compiler generate code to effectively use the 78K/0 Series, the target device.

A C source program that uses the extended functions of this C compiler will use functions that depend on the microprocessor, but is compatible at the C language level for porting to other microprocessors. Therefore, programs can be ported to other microprocessors by making simple revisions to the C source program that uses extended functions.

11.1 Macro Names

This C compiler has the two types of macro names; one indicates the name of the device series for the target device and the other indicates the device name. To output object code for the target device, these names are set by the compilation options or the device type in the C source.

For details on macro names, see **Section 9.9** "**Compiler Definition Macro Names**."

11.2 Keywords

This C compiler utilizes the following tokens as keywords for extended functions. The entire keyword is described in lowercase letters. If it contains uppercase letters, the word is no longer regarded as a keyword.

Table 11-1. Additional Keywords

Reserved Words	Application
callt	callt and _ _callt functions
_ _callt	
callf	callf and _ _callf functions
_ _callf	
sreg	sreg and _ _sreg variables
_ _sreg	
noauto	noauto Function
norec	norec and _ _leaf functions
_ _leaf	
bit	bit, boolean, and _ _boolean type variables
boolean	
_ _boolean	
_ _interrupt	Hardware interrupt
_ _interrupt_brk*****	Software interrupt
_ _banked1 to 15*****	Bank functions
_ _rtos_interrupt*****	RTOS handler

* To be compatible with CC78K0, these tokens are reserved words. However, they are not supported as functions.

(1) Functions

callt, noauto, and norec are qualifier attributes. These are described at the beginning of the function declaration. The syntax for a qualified declarator is

qualifier-attribute usual-declarator function-name

The settings for a qualifier attribute are limited to the following. (noauto and norec cannot be simultaneously specified.) Also, these declarations cannot be made in other functions. Qualifier attributes prefixed by '_ _' are enabled even when the -ZA option is specified.

- callt/_ _callt
- noauto
- norec/_ _leaf
- callt noauto
- callt norec/_ _leaf
- noauto callt
- norec/_ _leaf callt
- _ _interrupt

(2) Variables

- sreg and _ _sreg are specified in the same way as register in the C language. (For details on sreg, see **Section 11.4** "**(3) Using saddr space**.")
- bit, boolean, and _ _boolean types are specified in the same way as char or int type specifiers in the C language.

11.3 Memory

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

(1) Memory models

The 64-Kbyte model combines the code segment and the data segment.

(2) Register banks

- No register banks

(3) Memory space

This C compiler uses the memory space in the following way.

Table 11-2. Memory Space Use (for Normal Model)

Address		Application	Size (bytes)
00	40 to 7FH	CALLT table	64
FE	20 to D7H	sreg variable, boolean type variable	184
FE	D8 to E7H	register variables (Note 1)	16
FE	E8 to EFH	Arguments of norec function (Note 2)	8
FE	F0 to F7H	Automatic variables of norec function (Note 3)	8
FE	F8 to FFH	Arguments of runtime library (Note 4)	8
FF	00 to FFH	sfr variable	256

Table 11-3. Memory Space Use (for Static Model)

00	40 to 7FH	CALLT table	64
FE	20 to EFH	sreg variable, boolean type variable	208
FE	F0 to FFH	Common area (Note 5)	16
FF	00 to FFH	sfr variable	256

Notes 1. The areas not used as register variable areas are used as sreg variable and Boolean type variable areas.

2. If no register variables were used, areas not used as norec argument areas are used as sreg variable and Boolean variable areas.

3. If no register variables and norec arguments were used, areas not used as norec automatic variable areas are used as sreg variable and Boolean type variable areas.

4. If no register variable, norec arguments nor automatic variables were used, areas not used as run-time library argument areas are used as sreg variable and Boolean variable areas.

5. The area used by the compiler via the -SM option parameter is modified.

Moreover, when the register variable optimize option (-QR) is not specified, areas described in notes 1 to 3 are always used as sreg variable and Boolean variable areas.

11.4 Using Extended Functions

Each extended function is described in the following order.

Function	:	Describes the function implemented by the extended function.
Effect	:	Describes the effect obtained by the extended function.
Procedure	:	Describes how to use the extended function.
Limits	:	Describes the limits when an extended function is used.
Example	:	Illustrates an example using the extended function.
Description	:	Explains the example.
Compatibility	:	When a C source program developed using another C compiler is compiled by this C compiler, the compatibility of the C source program is described.

(1) callt [Function]

callt function	**callt/_ _callt**

[Function]

- The callt instruction stores the address of the called function in the area called the callt table. The function can be called in more compact code than by directly calling the function.
- When calling a function declared with callt (or _ _callt) (calling the callt function), the name used is the function name prefixed by a question mark (?). The callt instruction is used to call.
- The called function does not differ from an ordinary function.

[Effect]

- The object code can be reduced.

[Procedure]

- The callt or _ _callt attribute is added to the called function. (It is 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 */
            :
}
```

[Limits]

- The address of the function declared with callt or _ _callt is located in the callt table. However, since positioning in the callt table is performed during linking, the routine created when the callt table is used in the assembler source module uses symbols and is relocatable.
- The number of callt functions is checked during linking.
- callt table area: 40H to 7FH
- When the -ZA option is specified, _ _callt is enabled, and callt is disabled.

Table 11-4. Usage Limitations on the callt [Function]

	Option	Limit	Description
Counts per load module	No -QL	Max. 30	To use two callt table in an execution of multiplication or division prveessing.
Total counts in the linked module	No -QL	Max. 30	To use two callt table in an execution of multiplication or division prveessing.
Counts per load module	-QL/-QL specified	Max. 27	To use five callt tables
Total counts in the linked module	-QL/-QL specified	Max. 27	To use five callt tables
Counts per load module	-QL2 specified	Max. 13	To use 19 callt tables
Total counts in the linked module	-QL2 specified	Max. 13	To use 19 callt tables
Counts per load module	-QL3 specified	Cannot use	To use 32 callt tables
Total counts in the linked module	-QL3 specified	Cannot use	To use 32 callt tables

For details on bank functions, see **Section 11.4** “**(24) Bank functions**.”

callt function **callt**

[Example]

(C source)

```
============ cal.c ============   ============ ca2.c ============
_ _callt extern int tsub();

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

(Output object)

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

ca2 module
        PUBLIC _tsub                 ; Declaration
        PUBLIC ?tsub                 ;
@@CODE CSEG
_tsub:                               ; Function definition
               :
        Function body
               :

@@CALT      CSEG      CALLT0           ; Allocate to segment
?tsub:    DW    _tsub
```

[Description]

- The tsub( ) function that is called adds the callt attribute to store the address in the callt table.

[Compatibility]

<From another C compiler to this C compiler>

- If the callt or _ _callt keyword is not used, modifications are not required.
- If changing to a callt function, modify by following the procedure described earlier.

<From this C compiler to another C compiler>

- #define is used. For details, see **Section 11.5** “**Modifying the C Source**.”

(2) Register variables

Register variables	register

[Function]

- Register variables are allocated to the registers or saddr areas below in the order in which they were declared.

  Normal model : HL, saddr area [0FED8 to 0FEE7H]
  However, a register variable is allocated only if no stack frame exists in register HL.
  Static model : DE
  Allocation to the saddr area via register variable declaration is not performed.

- The registers or the saddr area are saved or restored in pre- and post-processing of a module with a register declaration.
- However, allocation to the saddr area is performed only when the -qr optimization option is specified (normal model only).

[Effect]

- Instructions for the registers and saddr area are designed to be shorter than memory instructions, to reduce the size of the object code, and to improve the execution speed.

[Procedure]

- The register class is declared by the storage-class specifier.

  register type-name variable-name

[Example]

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

[Limits]

- If register variables are used infrequently, the object code also increases (this depends on the scale and contents of the source).
- Register variable declarations can be used for char, int, short, long, float, double, long double, and pointers (long, float, double, and long double are valid for only normal model).

(For normal model)

- char uses half the area compared to the other types. long, float, double, and long double use twice the area.
- A pair of char type has byte boundaries. The other cases have word boundaries.
- The long, float, double, and long double types are not allocated to the HL register, but other types are allocated to it.
- Register variables of type int, short, or pointer in one function : A maximum of eight variables can be used. (The ninth and later variables are allocated to ordinary memory.)
- Register variables of type int, short, or pointer in a function that does not have a stack frame : A maximum of nine variables can be used. (The tenth and later variables are allocated to ordinary memory.)

(For static model)

- Char uses half the area campared to the other types.
- A maximum of 1 variable per function can be used for int, short, and pointers.
- The 2nd and later variables are allocated to ordinary memory.
- long, float, double, and long double are disabled.

Register variables **register**

[Example]

(C source)

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

(Compiler output object: normal model)

```
@@CODE CSEG
_main:
      push  hl
      movw  ax, _@KREG14
      push  ax
      movw  hl, #00H     ;0
      movw  ax, hl
      incw  ax
      movw  _@KREG14, ax ;j
      xch   a, x
      add   l, a
      xch   a, x
      addc  a, h
      movw  hl, ax
      pop   ax
      movw  _@KREG14, ax
      pop   hl
      ret
      END
```

Register variables **register**

(Compiler output object: static model)

```
@@CODE CSEG
_main:
      push  de
      movw  de, #00H;0
      movw  ax, #01H;1
      mov   !?L0003+1, a     ;j
      xch   a, x
      mov   !?L0003,a        ;j
      add   a, e
      xch   a, x
      addc  a, d
      movw  de, ax
      pop   de
      ret
      END
```

[Description]

- When using register variables, the storage class for variables becomes only the register class.

[Compatibility]

<From another C compiler to this C compiler>

- If the compiler supports register declarations, modifications are not needed.
- If you wish to use register variables, add the register declarations.

<From this C compiler to another C compiler>

- If the compiler supports register declarations, modifications are not needed.
- The number of register variables and which area to allocate them to depends on the compiler being used.

(3) Using saddr space

saddr area use	**sreg/_ _sreg**

(a) Use with sreg declarations

[Function]

- An object called an sreg variable that is declared with sreg or __sreg is automatically allocated to the saddr area (relocatable allocation).
- sreg variables in the C source are handled in the same way as ordinary variables.
- Each bit in an sreg variable of type char, short, int, or long automatically becomes a boolean type variable.
- An sreg variable that was declared without initial value has the initial value of zero.
- A variable declared by sreg is relocatable and allocated to the saddr area.
  Normal model : [0FE20 to 0FED7H]
  Static model : [0FE20 to 0FEEFH]
- The area that can be referenced by the sreg variables declared in assembler source is the saddr area [0FE20H to 0FEFFH].
  However, care must be taken because the compiler uses [0FED8 to 0FEFFH] (normal model) and [0FEF0 to 0FEFFH] (static model).

[Effect]

- Instructions for the saddr area are designed to be shorter than memory instructions, to reduce the size of the object code, and to improve the execution speed.

[Procedure]

- sreg and _ _sreg are declared in a module that defines the variables. This cannot be described in functions.

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

- The following declaration is made in a module that references variables. This can be described in a function.

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

saddr area use **sreg/__sreg**

[Limits]

- If sreg or_ _sreg is specified in the const type or a function, a warning message is output and the sreg and _ _sreg declaration is ignored.
- When -ZA is specified, only _ _sreg is enabled, and sreg is disabled.
- char uses half the area compared to other types. The long, float, double, and long double types use twice the area.
- A pair of char type has byte boundaries. The other cases have word boundaries.

(Noraml model)

- sreg or _ _sreg can not be declared in arguments of the function and automatic variables. (Enables when specifying the static model.)
- Variables that can be used in one load module for int, short, or pointer type: Maximum of 92 variables (when using the saddr area [FE20H to FED7H]).
  However, when a bit, boolean type variable, register variable, or norec/noauto function is used the maximum number of usable variables decreases.

(Static model)

- Variables that can be used in one load module for int, short or pointer type: Maximum of 104 variables (when using the saddr area [FE20H to FEEFH]).
  However, when a bit or Boolean type variables, or a common area, is used the maximum number of usable variables decreases.

[Example]

(C source)

```
============ sal.c ==============

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

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

saddr area use **sreg/__sreg**

(Assembler source)

(In this case, the user creates the definition code for the sreg variable. However, if the extern declaration is not included, this C compiler outputs the following code. This ORG pseudo-instruction is not output.)

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

@@DATS DSEG SADDRP              ; Allocated to segments.
      ORG   0FE20H              ;
_hsmm0:     DS     (2)          ;
_hsmm1:     DS     (2)          ;
_hsptr:     DS     (2)          ;
```

(Compiler output object)

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
```

[Description]

When using sreg variables, only the sreg or _ _sreg attribute is added to the variables.

[Compatibility]

<From another C compiler to this C compiler>

- If the sreg/_ _sreg keyword is not used, modifications are not needed.
- When modifying the sreg function, modifications must conform to Procedure above.

<From this C compiler to another C compiler>

- #define is used. For details, see Section **11.5 "Modifying the C source."** By making this change, the sreg function is treated as an ordinary variable.)

saddr area use **-QD**

(b) Use with sreg automatic allocation options for external variables or external static variables

[Function]

- Regardless of the presence or absence of the sreg declaration, external variables or external static variables (except for const type) are automatically allocated in the saddr area.
- The maximum width of the variables allocated by the n value has the following differences.

  (a) When 1 - Variable of type char or unsigned char
  (b) When 2 - Variables when 1 and variables of type short, unsigned short, int, unsigned int, enum, or pointer
  (c) When 4 - Variables when 2 and variables of type long, unsigned long, float, double, or long double
  (d) Default - All variables (including the structure, union, array in only this case)

- A variable declared with sreg is allocated to the saddr area regardless of the above specifications.
- The processing allocates the variables referenced by the extern declaration to the saddr area in accordance with the above description.
- Variables allocated in the saddr area by this option are handled in the same way as sreg variables and have the functions and limits described in (1).

[Specification]

The -qd[n] option, where n is 1, 2, or 4, is specified.

[Limits]

Modules for which a different n is specified cannot be linked with the -qd [n] option.

saddr area use **-QS**

(C) Use with sreg automatic allocation options for internal static variables

[Function]

- Regardless of the presence or absence of the sreg declaration, internal static variables (excluding const type) are automatically allocated in the saddr area.
- The maximum width of the variables allocated by the n value differs as follows.

  (a) When 1 - Variable of type char or unsigned char
  (b) When 2 - Variables when 1 and variables of type short, unsigned short, int, unsigned int, enum, or pointer
  (c) When 4 - Variables when 2 and variables of type long, unsigned long, float, double, or long double
  (d) Default - All variables (including the structure, union, array)

- A variable declared with sreg is allocated to the saddr area regardless of the above specifications.
- Using this option, variables allocated to the saddr area are treated in the same way as sreg variables, and have the functions and limits described in (1).
- Modules for which a different n is specified cannot be linked with the -qs[n] option.

[Specification]

The -qs[n] option, where n is 1, 2, or 4, is specified.

saddr area use **-QK**

(d) Use with sreg automatic allocation options for arguments or automatic variables

[Function]

Regardless of the presence or absence of the sreg declaration, arguments and automatic variables (except cost type) are automatically allocated in the saddr area.

- The maximum width of the variables allocated by the n value differs as follows.

  (a) When 1 - Variable of type char or unsigned char
  (b) When 2 - Variables when 1 and variables of type short, unsigned short, int, unsigned int, enum, or pointer
  (c) When 4 - Variables when 2 and variables of type long, unsigned long, float, double, or long double
  (d) Default - All variables (including the structure, union, array)

- A variable declared with sreg is allocated to the saddr area regardless of the above specifications.
- Using this option, variables allocated to the saddr area are treated in the same way as sreg variables, and have the functions and limits described in (1).
- Modules for which a different n is specified cannot be linked with the -qs[n] option.

[Specification]

The -qk[n] option, where n is 1, 2, or 4, is specified.

[Limits]

- Only the normal model is supported. When the -SM (Static model specification) option is not specified, a warning message is output, and automatic allocation to the saddr is not performed.
- Arguments or variables declared with a register variable are not allocated to the saddr area.
- When the -qv option is specified, allocation using this option is prioritized, therefore 2-byte argument/automatic variables are the only things that are allocated to register DE.

saddr area use **-QK**

[Example]

(C source)

```
sub (int hsmarg)
{
        int hsmauto;
        hsmauto = hsmarg;
}
```

(Compiler output object)

```
@@ DATS     DSEG     SADDRP
 ?L0003:    DS       (2)
 ?L0004:    DS       (2)

@@ CODE     CSEG
 _sub:
       movw  ?L0003, ax
       movw  ax, ?L0003      ;hsmarg
       movw  ?L0004, ax      ;hsmauto
       ret
```

(4) Using the sfr area

sfr area use	**sfr**

[Function]

- The sfr area is a register group area for the allocation of special functions for mode registers or control registers in peripheral hardware for the 78K/0S Series.
- Declaring the use of the sfr name can describe the operations on the sfr area at the C source level.
- The sfr variable is an external variable that is not initialized (undefined).
- Writing of a read-only sfr variable is checked.
- Reading of a write-only sfr variable is checked.
- If incorrect data is assigned to an sfr variable, a compiler error results.
- sfr names that can be used are allocated to [0FF00H to 0FFFFH].

[Effect]

- Operations in the sfr area can be described at the C source level.
- Instructions for sfr are designed to be shorter than memory instructions, to reduce the size of the object code, and to improve the execution speed.

[Procedure]

- #pragma instructions are used to declare the use of sfr names in C source. (The sfr keyword can be described in lowercase or uppercase letters).

  ```
  #pragma sfr
  ```

- The #pragma sfr is described at the beginning of the C source. However, if the #pragma PC(type) is specified, later #pragma sfr is described.
  The following items can be described before the #pragma sfr.

  - Comments
  - Preprocessing directives that do not generate variable or function definitions or references

- An sfr name for a device is described unchanged in the C source. At this time, the sfr name does not have to be declared.

[Limits]

- An sfr name is described in uppercase letters. A name in lowercase letters is treated as an ordinary variable. However, if the symbol name case specification option is described during compilation, uppercase and lowercase letters are not distinguished, and all letters are regarded as uppercase letters. In this case, even when the name is described in lowercase letters, it is treated as an sfr name. For details on the compilation options, see the “**CC78K0 Series C Compiler User’s Manual, Operation**.”

sfr area use **sfr**

[Example]

(C source)

```
#ifdef _ _K0S_ _
          #pragma sfr
#endif

void main()
{
        P0 -= RXB;
         :
}
```

(Compiler output object)
Code related to declarations is not output. The following code is output in the function.

```
_main:
        mov     a, P0
        sub     a, RXB
        mov     P0, a
        ret
```

[Description]

- In this example, the use of sfr variables is indicated by “#pragma sfr.” In the program, sfr names are used to use special function registers like P0 (port 0), the in-service priority register, and TXB.

[Compatibility]

<From another C compiler to this C compiler>

- If the part does not depend on the device or the compiler, modifications are not needed.

<From this C compiler to another C compiler>

- The “#pragma sfr” statements are deleted, or variable declarations are added for sfr variables separated by “#ifdef”. This is illustrated in the example.

```
#ifdef _ _K0S_ _
          #pragma sfr
#endif

/* Variable declarations */
```

- For a device having sfr or a function replacing it, a special library must be created to access that area.

(5) noauto function

noauto function **noauto**

[Function]

- A function that does not use automatic variables can be a noauto function declared with noauto. When the function is a noauto function, code for pre- and postprocessing (creating the stack frame) is not output.
- If passed arguments can be used, they are stored in the registers or saddr area [0FEEC to 0FEEFH] for register variables.
- Inside the noauto function, arguments passed via a register or stack are copied to a register (because the registers on the noauto function side call side and the definition side differ). Moreover, the register allocating arguments is saved and restored on the function definition side.
- The first argument is stored to register HL. However, long/float/double/long double are not stored to HL but allocated to other arguments.
- Arguments not allocated to registers are store to the saddr area for register variables.
  These arguments are stored in ascending order in the description order. (See _@KREG12 to 15 in Appendix A “SADDR SPACE LABEL SUMMARY”)
- Automatic variables can be used only when all automatic variables have been allocated to registers and saddr areas for register variables left over after argument allocation.
  Registers to which automatic variables are allocated are saved and restored on the function definition side.
- If the -SM (Static Model specification) option is specified, a warning message is output only at the line where noauto is described for the first time, and all noauto functions are treated as ordinary functions.
- Arguments are passed to a noauto function in the same way as in the case of ordinary functions.

Table 11-5. noauto Function Passing List (noauto Function Calling Side)

Argument Type	1st Argument	2nd Argument
1-, 2-byte integer, pointer	Passed on AX	Passed on stack
4-byte integer	Passed on AX and BC	Passed on stack
Floating-point number	Passed on AX and BC	Passed on stack
Other	Passed on stack	Passed on stack

* 1- to 4-byte integers include structures and unions

Table 11-6. noauto Function Interface (noauto Function Definition Side)

Automatic Variable Argument	Allocation Order	Storage Method
Automatic variables	Same as arguments	Allocates arguments to the registers and saddr area. The remainder is stored as follows (only when the -QR option is specified, arguments are allocated to the saddr area): HL (only when the stack frame is none) saddr area (_@KREG12 to 15[0FEE4 to E7H])
char type arguments int, short, and enum type arguments	In order of L and H HL	HL (only when the stack frame is none)
char type arguments int, short, and enum type arguments long, float, and double types Arguments	In ascending order of _@KREG12 to 15 _@KREG12, 13 $\rightarrow$ 14, 15 _@KREG12 to 13 (low-order) 14 to 15 (high-order)	saddr area(_@KREG12 to 15[0FEE4 to E7H]) (Allocated only when the -QR option is specified.)

[Effect]

- The object code is reduced and the execution speed is improved.

[Procedure]

- The noauto attribute is declared during function declaration.

  noauto type-name function-name

noauto function **noauto**

[Limits]

- When -ZA is specified, noauto is disabled.
- The types and number of the arguments passed to the noauto function are restricted.
  The types of the arguments that can be used in the noauto function are

  - 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

- However, long/signed long/unsigned long and float/double/long double are not allocated to the HL register but to other arguments.
- The arguments and automatic variables that can be used have a maximum of six bytes.
- These limits are checked during compilation.
- If register is declared with an argument, the register declaration is ignored.

[Example]

(C source)

```
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 function **noauto**

(Compiler output object)

```
        @@CODE CSEG
_main:
        mov     a,!?L0005     ; kk
        xch     a,x
        mov     a,!?L0005+1   ; kk
        push    ax
        mov     a,!?L0004     ; jj
        xch     a,x
        mov     a,!?L0004+1   ; jj
        push    ax
        mov     a,!?L0003     ; ii
        xch     !a,x
        mov     a,!?L0003+1   ; ii
        call    !_nfunc       ; Call nfunc function.
        pop     ax
        pop     ax
        movw    ax,bc
        mov     !_l+1,a       ; Assign the return value to the extgernal variable 1.
        xch     a,x
        mov     !_l,a
        ret

_nfunc:
        push    hl            ;Save HL.
        xch     a,x
        xch     a,_@KREG12    ;Set the argument a to _@KREG12.
        xch     a,x
        xch     a,_@KREG13
        push    ax            ;Save _@KREG12.
        movw    ax,_@KREG14
        push    ax            ;Save _@KREG14.
        movw    ax,sp
        movw    hl,ax
        mov     a,[hl+10]
        xch     a,x
        mov     a,[hl+11]
        movw    _@KREG14,ax   ;Set the argument c to _@KREG14.
        mov     a,[hl+8]
        xch     a,x
        mov     a,[hl+9]
        movw    hl,ax         ;Set the argument b to HL.
        movw    ax,hl
        xch     a,x
        add     a,_@KREG12    ;a
        xch     a,x
        addc    a,_@KREG13    ;a
        xch     a,x
        add     a,_@KREG14    ;c
```

noauto function **noauto**

(Compiler output object) - continued

```
        xch     a,x
        addc    a,_@KREG15    ;c      Add b(HL) and c(_@KREG14) to a(_@KREG 12).
                                      Assign the computation result to the external
                                      variable m.
        mov     !_m+1,a       ;
        xch     a,x
        mov     !_m,a
        xch     a,x
        movw    bc,ax         ;Retrun the contents of external variable m.
        pop     ax
        movw    _@KREG14,ax   ;Restore _@KREG 14.
        pop     ax
        movw    _@KREG12,ax   ;Restore _@KREG12.
        pop     hl            ;Restore HL.
        ret
        END
```

[Description]

- In this example, the noauto attribute is added to the header.
  noauto is declared so that the stack frame is not generated.

[Compatibility]

<From another C compiler to this C compiler>

- If the noauto keyword is not used, modifications are not needed.
- When modifying the noauto function, modifications must conform to **Procedure** above.

<From this C compiler to another C compiler>

- #define is used. For details, see **Section 11.5** “**Modifying the C Source**.”

(6) norec [Function]

norec function **norec**

[Function]

- A function that does not call another function can be a norec function.
- In a norec function, code for pre- and postprocessing of the function (creating the stack frame) is not output.
- The arguments are stored in the registers or the saddr area (-@NRARG0 to 3 [0FEE8 to 0FEEFH]), and the norec function is called.
- Automatic variables are allocated to the saddr area (-@NRAT00 to 07 [0FEF0 to 0FEF7H]), and are similar to register variables.
- Allocation is to the saddr area only when the -QO option is not specified during compilation.
- If the arguments do not have type long, float, double, or long double, the first argument is stored in the ax register, the second argument in the de register, and the third and later arguments in ascending order in the saddr area.
  If the arguments are long, float, double, or long double type, the first and subsequent arguments are stored in ascending order in the saddr area. However, the ax and de registers only store one argument regardless of the type of the argument.
- If the argument stored in ax is at the beginning of a norec function and no argument is stored in de, it is copied to de. If an argument is stored in de, it is copied to _@RTATG6 and 7.
- If the automatic variables do not have type long, float, double, or long double, any ones remaining after argument allocation are declared in order and stored in de, _@RTARG6, _@RTARG7, _@NRARG0, _@NRARG1,....
  If the automatic variables have type long, float, double, or long double, any ones remaining after argument allocation are declared in order and stored in _@NRARG0, _@NRARG1,....
  The remaining variables are declared in order and stored in the saddr area.
  (For details on _@RTARG6, _@RTARG7, _@NRARG0, _@NRARG1,..., see **Appendix A** “**SADDR SPACE LABEL SUMMARY**.”)

Table 11-7. norec Function Arguments Pass List (norec function calling side)

Argument type	1st argument	2nd argument or more
1-byte, 2-byte integer pointer	Passed on AX (DE on receive side)	Passed on DE (_@RTARG6 to 7 on receive side) Passed on _@NRARG0 to 3[0FEE8 to EFH]
4-byte integer	Passed on AX and DE (DE and _@RTARG6 to 7 on receive side)	Passed on _@NRARG0 to 3 [0FEE8 to EFH]
Floating-point number	Passed on AX and DE.	Passed on _@NRARG0 to 3 [0FEE8 to EFH]

norec function **norec**

Table 11-8. norec Function Interface

Automatic Variables and Functions	Storage Method

Automatic Variables	Arguments are allocated to the registers and saddr. If any remain, they are stored as follows.
	[If not long, float, double, or long double]
	`DE`
	`saddr area`
	`(_@RTARG6, 7 [0FEFE to FFH], _@NRARG0 to 3 [0FEE8H to F7H])`
	[If long, float, double, or long double]
	`saddr area`
	`(_@NRARG0,1, … [0FEE8H to F7H])`

First argument	AX (allocation of only one argument if not long, float, double, or long double)
Second argument	DE (allocation of only one argument if not long, float, double, or long double)
Third argument	saddr area (_@NRARG0 to 3 [0FEE8H to 0FEEFH])

[Effect]

- The object code can be reduced, and the execution speed of program improved.

[Procedure]

- The norec attribute is specified when the function is declared.

  norec type-name function-name

- _ _leaf can be described instead of norec.

[Limits]

- Other functions cannot be called from a norec function.
- The size and number of arguments and automatic variables in a norec function are restricted.
  When -ZA is specified, norec is disabled, and only _ _leaf is enabled.
- When the-SM option is specified, the worning message appears first for only lines which norec is described. All norec functions are processed as a normal function.
- The usable automatic variables are:
  [When -QR option is specified]
  - Maximum of 4 bytes (long, float, double, or long double cannot be used.)

  [When -QR option is not specified]
  - Maximum of 20 bytes
    (Maximum of 16 bytes for long, float, double,or long double)

norec function **norec**

- The usable arguments are

  [When -QR option is specified]
  Maximum of two variables (long, float, double, or long double cannot be used)
  [When -QR option is not specified]
  Maximum of six variables
  (long, float, double, or long double contains up to two variables.)

- These argument types can be used in the norec Function

  - 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

- If a pair has type char, signed char, or unsigned char, they are consecutively allocated to the saddr area. If other types are contiguous, the arguments are allocated with two byte alignment.

[Example]

(C source)

```
norec char rout (int a, int b, int c);

int i, j;
void main()
{
         int k, l, m;
         i = l + rout (k, l, m) + ++k ;
}

norec char rout (int a, int b, int c)
{
         int x, y;
         return (x + a<<2));
}
```

norec function **norec**

```
(Compiler output object)                     (When the -QR option is specified)

             :
       EXTRN   _@NRARG0        ; Reference the saddr area being used
       EXTRN   _@NRARG6        ;
       EXTRN   _@NRARG1        ;
             :
@@CODE       CSEG
_main:
             :
       movw    ax,sp
       movw    hl,ax
       mov     a,[hl]    ;m
       xch     a,x
       mov     a,[hl+1]  ;m
       movw    _@NRARG0,ax     :Save the argument in the saddr area.
       mov     a,[hl+2]  ;l
       xch     a,x
       mov     a,[hl+3]  ;l
       movw    de,ax           :Save the argument in the saddr area.
       mov     a,[hl+4]  ;k
       xch     a,x
       mov     a,[hl+5]  ;k    :Save the argument in the ax register.
       call    !_rout          ;Call the norec function.
                :
       ret
_rout:
       movw    _@RTARG6,ax     ;Receive an argument in the saddr area.
       mov     c,#02H    ;2
       xch     a,x
       add     a,a
       xch     a,x
       rolc    a,1
       dbnz    c,$$-5
       xch     a,x
       add     a,_@NRARG1      ;x     Use the automatic variable in the saddr area.
       xch     a,x
       addc    a,_@NRARG1+1    ;x
       movw    bc,ax
?L0005:
       ret
       END
```

norec function **norec**

[Description]

The norec attribute is added to indicate that the function is a norec function in the rout function definition.

[Compatibility]

<From another C compiler to this C compiler>

- If the norec keyword is not used, modifications are not needed.
- If changing to a norec function, modifications must conform to **Procedure** above.

<From this C compiler to another C compiler>

- #define is used. For details, see **Section 11.5** “**Modifying the C Source**.”

(7) bit type variables

bit type variables	**bit**
boolean type variable	**boolean**
	_ _boolean

[Function]

- bit and boolean type variables define one bit data.
- bit and boolean type variables are handled in the same way as uninitialized external variables (undefined).
- The compiler for these bit variables can output the following bit-manipulation instructions.

  SET1, CLR1, NOT1, BT, and BF instructions are output.

[Effect]

- Bit accessing the saddr and sfr areas is possible in programming in the assembler source level in C descriptions.

[Procedure]

- bit and boolean type declarations are made in modules that use bit or boolean type variables.
- A _ _boolean description can be used instead of bit.

```
bit variable-name
boolean variable-name
_ _boolean variable-name
```

- extern bit (boolean) declarations are made in modules that reference bit or boolean type variables.

```
extern bit variable-name
extern boolean variable-name
extern _ _boolean variable-name
```

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

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

[Limits]

- Computations on pairs of bit or boolean type variables use the carrier flag. Therefore, the contents of the carrier flag cannot be guaranteed between statements.
- Arrays cannot be defined or referenced.
- These variables cannot be used as members in structures or unions.
- These types cannot be used as the types of the function arguments.
- They cannot be declared with initial values.
- If written with the const declaration, the const declaration is ignored.
- For normal model, bit type variables cannot be used as a type of automatic variable. (Enable for normal model).
- Computations with constants based on the following operators can only be on 0's or 1's.

bit type variables	**bit**
boolean type variable	**boolean**
	_ _boolean

Table 11-9. Operators That Act Only on the Constants 0 or 1 (When using bit type variables)

Class	Operator	Class	Operator
Assignment	=		
Bitwise AND	.&, &=	Bitwise OR	|, |=
Bitwise XOR	^, ^=		
Logical AND	&&	Logical OR	| |
Equal	==	Not equal	!=

- A maximum of 1,720 variables can be used in one load module for bit type variables (when using the saddr area [0FE20 to D7H]
  However, if sreg variables are used and -QD (sreg allocation option for external variables or static variables), -QS (sreg allocation option for internal static variables), and -QK (sreg allocation option for arguments or automatic variables) are specified, the number of variables that can be used decreases.
- *, & (pointer reference, address reference), and the sizeof operations cannot be performed.
- When the -ZA option is specified, only _ _boolean is valid.

[Example]

(C source)

```
#define ON 1
#define OFF 0
void testb (void);
void chgb (void);
bit data1;
boolean data2;

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

bit type variables	**bit**
boolean type variable	**boolean**
	_ _boolean

(Assembler source)

(The definition code for bit type variables written by the user is illustrated. However, if the extern declaration is not added, the compiler outputs the following code. At this time, the ORG pseudo-instruction is not output.)

```
PUBLIC _data1        ; Declaration
PUBLIC _data2

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

(Compiler output object)

The following codes are output in the function.

```
      set1    _data1           (Initialization)
      clr1    _data2           (Initialization)
?L0003:
      bf      _data1, $?L0004  (Decision)
      set1    CY               (Assignment)
      bt      _data2,$?L0005
      clr1    CY               (Assignment)
?L0005:
      bnc     $?L0006
      set1    _data1
      br      $?L0007
?L0006:
      clr1    _data1
?L0007:
      call    !_testb
      br      $?L0003
?L0004:
      bf      _data1,$?L0008   (Logical AND expression)
      bf      _data2,$?L0008   (Logical AND expression)
      call    !_chgb
?L0008:
?L0009:
      ret
      END
```

bit type variables	**bit**
boolean type variable	**boolean**
	_ _boolean

[Description]

The bit type is specified for data1 and data2 to indicate that they are bit type variables.

[Compatibility]

<From another C compiler to this C compiler>

- If the bit and boolean keywords are not used, modifications are not needed.
- If changing to bit and boolean variables, the modifications must conform to **Procedure** above.

<From this C compiler to another C compiler>

- #define is used. For details, see **Section 11.5** "**Modifying the C Source**." (By making this change, bit and boolean type variables are treated as ordinary variables.)

(8) ASM statement

ASM statements	**#asm** **_ _asm**	**#endasm**

[Function]

(1) #asm - #endasm

- Assembler source described by the user is embedded in the assembler source file output by this C compiler.
- The #asm and #endasm lines are not output.

(2) _ _asm

- Assembly code is described in string literals.
  Assembly instructions are output and inserted in the assembler source.

[Effect]

- Global variables in the C source can be manipulated in the assembler source.
- Functions that cannot be described in the C source can be implemented.
- By optimizing the assembler source output by the C compiler by hand and embedding it in the C source, a highly efficient object is obtained.

[Procedure]

(1) #asm - #endasm

- #asm indicates the beginning and #endasm indicates the end of the assembler source. The assembler source is described between #asm and #endasm.

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

(2) _ _asm

- The use of _ _asm is declared by specifying #pragma asm at the beginning of the module that describes the ASM statement. (The keyword following #pragma can be in lowercase or uppercase letters.)
- The following items can be described before #pragma asm.

  - Comments
  - Other #pragma directives
  - Preprocessing directives that do not generate the definitions or references to variables or functions

- The format in the C source is

  _ _asm(string-literal);

- The description of the string literals conforms to ANSI and can describe escape sequences (\n: new line, \t: tab, etc.) or \ for line continuation and concatenating strings.

ASM statements	**#asm** **_ _asm**	**#endasm**

[Limits]

- "#asm - #endasm" and _ _asm can only be described in a C source function. Consequently, the assembler source is output to CSEG in the @@CODE segment.
- #asm cannot be nested.
- When the ASM statement is used, the object module file is not generated, but the assembler source file is generated.
- _ _asm can only be described in lowercase letters (when the -ca option is specified). If lowercase and lowercase letters are mixed in the description, the function is regarded as a user function.
- When the -ZA option is specified, only _ _asm is valid.

[Example]

(1) #asm - #endasm

(C source)

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

(Compiler output object)

The assembler source described by the user is output to the assembler source file.

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

[Description]

- Assembler source is described between #asm and #endasm and output to the assembler source file.

ASM statements	#asm _ _asm	#endasm

(2) _ _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]

- In C compilers that support #asm, modifications conform to the format specified for that C compiler.
- If the target device is different, the assembler source is modified.

(9) Kanji

Kanji	**/* Kanji */**

[Function]

- Kanji can be described in comments in the C source.
- The kanji in comments is treated as a comment and is not a compilation target.
- The kanji code used in the comments can be selected by options or environment variables. If no options are specified, the setting in the LANG78K environment variable is used. (Setting the LANG78K variable to NONE is interpreted as no kanji code is present.) The defaults are given below.
- If both options and the LANG78K environment variable are set, the option settings are enabled.

<When MS-DOS based>
- If the LANG78K environment variable is not set, SJIS is set.

<When PC DOS based>
- If the LANG78K environment variable is not set, NONE is set.

<When HP or NEWS based>
- If the LANG78K environment variable is not set, SJIS is set.

<When SUN4 based>
- If the LANG78K environment variable is not set, EUC is set.

[Effect]

- Comments can be written to ease understanding and simplify C source management.

[Procedure]

- Either the -ZS, -ZE, and -ZN option is specified.

Table 11-10. Kanji Options

Option	Description
-ZS	SJIS (shift JIS code)
-ZE	EUC (EUC code)
-ZN	NONE (no kanji code)

- The LANG78K environment variable is set to EUC, SJIS, or NONE.
- EUC, SJIS, or NONE can be described in lowercase or uppercase letters.
- Kanji is described in the comments of the C source (EUC code when the LANG78K environment variable is set to EUC and shift JIS code when set to SJIS).

[Limits]

- If kanji is not supported by the operating system, kanji cannot be used.
- Kanji can only be described in comments.

Kanji /* Kanji */

[Example]

(C source)

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

(Compiler output object)

If C source is output to the assembler source, the kanji in the comments and kanji-related information are output.

```
$KANJICODE SJIS
       :
; line 1       void main()      /* main function */
; line 2       {
; line 3                        /* Comment */
```

[Description]

- Kanji can only be used in comments in the C source.

[Compatibility]

<From another C compiler to this C compiler>

- The source must be modified if kanji is located outside of comments (outside of /*...*/).
- If the kanji code is different, the kanji code must be converted.

<From this C compiler to another C compiler>

- The C source is not modified for kanji in comments for a C compiler that allows kanji to be written in the comments.
- The kanji in the C source must be deleted for a C compiler that does not allow kanji to be written in the comments.

(10) Interrupt functions

Interrupt functions	#pragma vect/interrupt

[Function]

- The address of the described function name is registered in the interrupt vector table for the specified interrupt request name.
- In an interrupt function, the code for saving and restoring the following items (unless used in an ASM statement) is output at the beginning and the end of the interrupt function (after the code for a register bank specification).

  (1) Register
  (2) saddr area for the run-time library
  (3) saddr area for register variables
  (4) saddr area for auto variables or arguments of norec functions (whether used or not)

  However, the areas for saving and restoring differ with the specification and status of the interrupt function.

- When the specification is not changed, code is not output to change the register banks, to save or restore registers, and to save or restore the saddr area.
- However, when the specification is not changed, if there is a function call in an interrupt function, the entire area is saved and restored regardless of whether is used or not.

<Normal model>

- f the -QR option is not specified during compiling, the saddr area for register variables and the saddr area for norec function arguments/auto variables are not used, and therefore save and restore code is not output.
- If the size of the entire save code is smaller, the entire save code is output.
- The above save/restore area information is summarized in the table below.

Interrupt functions **#pragma vect/interrupt**

Table 11-11. Save or Restore Area When Using Interrupt Functions (Normal Model)

Save/Restore Area	NO BANK	Function Call		No Function Call	
		No -QR	-QR	No -QR	-QR
Register used	—	—	—	√	√
All registers	—	√	√	—	—
saddr area for the run-time libraries being used	—	—	—	√	√
saddr area for all run-time libraries	—	√	√	—	—
saddr area for the register variables being used	—	—	√	—	√
Entire saddr area for norec function arguments or auto variables	—	—	√	—	—

Stack :	Stack use specification	√ :	Save
RBn :	Register bank specification	– :	Do not save

***** When there are ASM statements in the interrupt function and the reserved area for the compiler and registers are used (areas in the above table), the user is responsible for saving the areas.

<Static model>

- If the -SM option is specified during compiling, the saddr area for register variables and the saddr area for norec function arguments/auto variables, as well as saddr area for time libraries are not saved, and therefore only the registers' save and restore code is output, and the saddr area's save and restore code is not output. However, if leafwork1 to 16 were specified, the code that saves and restores the specified number of bytes from the upper address of the common area is output at the start and end of the interrupt function. (For details, see "Static model" in this chapter.)

[Effect]

- Interrupt functions can be described in the C source level.
- Since the interrupt request name is identified, the address of the vector table does not have to be required.

Interrupt functions **#pragma vect/interrupt**

[Procedure]

- Saving or restoring of interrupt request name, function name, register, as well as the saddr area that is used and the common area are specified by #pragma directives.
  (For the interrupt request name, see the user’s manual for the target device to be used.)
- Of the keywords described after #pragma, the interrupt request name must be entered as uppercase characters. Other keywords can be entered using both upper and lowercase characters.
- When the #pragma PC(type) is described, this #pragma directive is described later. The following items can be described before a #pragma directive.

  - Comments
  - Preprocessing directives that do not generate variable or function definitions or references

```
             vect
#pragma Δ {         } Δ interrupt-request-name Δ function-name Δ
           interrupt

                                                    stack-usage-specification
                                               [{    no-change-specification    }]
                                                   register-bank-specification
```

stack-usage-specification	:	STACK (default)
no-change-specification	:	NOBANK
register-bank-specification	:	leafwork 1 to 16

Interrupt functions **#pragma vect/interrupt**

[Limits]

- The interrupt request name must be entered in upper-case letters.
- Saving/storing of the common area can be specified only during specification of the static model (-SM), and cannot be specified in the normal model. If leafwork1 to 16 is specified when the -SM option is not specified, a warning is output and specification of the common area save/restore is ignored.
- Redundant checking is performed on the interrupt request names only in one module unit.
- The register contents are rewritten when multiple interrupts (same or different interrupt) are generated during vectored interrupt servicing by the contents of a priority specification flag register or an interrupt mask register, or due to the register bank specification or no-change specification. Sometimes they become incompatible, but the compiler cannot check this.
- An interrupt function cannot specify callt, noauto, norec, _ _callt, _ _leaf.
- Even when an ASM statement is in an interrupt function, all of the save code is not output. Consequently, when compiler reserved area in the ASM statement is used in an interrupt function, or when calling a function in the ASM statement, the user must perform the save.
- If the function which executes the no-change specification with the #pragma vect or #pragma interrupt specification, or specifies the common area saving/restoring is not defined in the same module, a warning is output and the save destination specification is ignored.
- If the common area is saved with the leafwork specification, the number of bytes to be specified must be set to the maximum bytes of the common areaallocated with the -SM option specification in all modules.

Interrupt functions **#pragma vect/interrupt**

[Example]

(C source : with common area saving/restoring specification)

```
        #pragma interrupt INTP0 inter leafwork4

        void inter()
        {
                  func();         /* Interrput processing for the port0 input*/
        }
```

(Compiler output object)

```
        EXTRN   _@KREG12
        EXTRN   _@KREG14
        EXTRN   _func
        PUBLIC  _inter
                :

@@CODE          CSEG
_inter:
        push    ax              ┐
        push    bc              │   Saving of register
        push    de              │
        push    hl              ┘
        movw    ax,_@KREG12     ┐
        push    ax              │   Saving of common area
        movw    ax,_@KREG14     │
        push    ax              ┘
        call    !_func
        pop     ax              ┐
        movw    _@KREG14,ax     │   Restoring of the saddr area compiler uses
        pop     ax              │
        movw    _@KREG12,ax     ┘
        pop     hl              ┐
        pop     de              │   Stack pointer returns to the begining
        pop     bc              │
        pop     ax              ┘
        reti

@@VECT06        CSEG  AT    0006H
        DW      _inter
        END
```

Interrupt functions **#pragma vect/interrupt**

[Compatibility]

<From another C compiler to this C compiler>

- If interrupt functions are not used, modifications are not needed.
- If changing to interrupt functions, the modifications must conform to the **Procedure** above.

<From this C compiler to another C compiler>

- If the #pragma vect or #pragma interrupt specification is deleted, the function is treated as an ordinary function.
- If using as an interrupt function, the modification must follow the specifications of each compiler.

(11) Interrupt function qualifiers (_ _interrupt)

Interrupt function qualifiers	Interrupt function qualifiers

[Function]

- By declaring a function with the _ _interrupt qualifier, that function is regarded as a hardware interrupt function and is returned by the RETI return instruction for hardware interrupt functions.
- A function declared with this qualifier is regarded as a hardware or software interrupt function. The area used as the operating area of the compiler in (1) to (3) below is saved and restored on the stack.
  However, when a function call is described in this function, the entire area is saved on the stack.

  (1) Register
  (2) saddr area for run-time libraries
  (3) saddr area for register variables
  (4) saddr area for norec function arguments or auto variables (whether used or not)

  If the -QR option is specified when compiling, since areas (3) and (4) are not used, save and restore codes are not output.
  When the -SM option is specified, since areas (2), (3) and (4) are not used, save and restore codes are not output.

[Effect]

- By declaring with this qualifier, the vector table settings and interrupt function definitions can be described in separate files.

[Procedure]

- _ _interrupt is added to the qualifier for an interrupt function.

  _ _interrupt void func(){ processing }

[Limits]

_ _ interrupt_brk is not supported in CC78K0 because of no software interruptions.
For a position where the _ _ interrupt_brk keyword appears first, a warning message is output and the keyword is ignored and processed as a normal function.

- An interrupt function cannot specify callt, noauto, norec, _ _callt, _ _leaf.

Interrupt function qualifiers **Interrupt function qualifiers**

[Warnings]

- The vector address is not set only when this qualifier is declared. The vector address setting must be separately performed by a #pragma directive or assembler description.
- The save destination of the saddr space and the registers becomes the stack.
- Even when the vector address is set or the save destination is changed by #pragma vect (or interrupt), the change in the save destination is ignored if there is no function definition in the same file. The default is the stack.
- When the interrupt function is defined in the same file as the #pragma vect (or interrupt) specification, the function name specified by the #pragma vect (or interrupt) is determined to be an interrupt function even when no qualifier is described. (For details on the #pragma vect, see **Procedure** in Section **(10)** "**Interrupt functions**.")

[Example]

The declarations and definitions of a function have the following formats.

```
_ _interrupt void func();                    /* Prototype declaration */
_ _interrupt_brk void func();                /* Prototype declaration */
_ _interrupt void func(){ Processing };      /* Function body */
_ _interrupt_brk void func(){ Processing };  /* Function body */
```

[Compatibility]

<From another C compiler to this C compiler>

- If interrupt functions are not supported, no modifications are needed.
- If you wish to change to interrupt functions, changes must conform to the **Procedure** described above.

<From this C compiler to another C compiler>

- #define is used. It can be handled as an ordinary function.
- If using as an interrupt function, the changes must conform to the compiler specifications.

(12) Interrupt functions

Interrupt function qualifiers	Interrupt function qualifiers

[Function]

- DI and EI code is output to the object and the object file is created.
- If there are no #pragma directives, DI( ) and EI( ) are regarded as ordinary functions.
- If "DI( );" is described at the beginning of the function (except for automatic variable declarations, comments, or preprocessing directives), DI code is output before the function's preprocessing (immediately after the function name label).
- If DI code is output after the function's preprocessing, a new block is opened, delimited by '{', before describing "DI( );".
- If "EI( );" is described last in the function (excluding comments or preprocessing directives), EI code is output after the function's postprocessing (immediately before RET code).
- If EI code is output before the function's postprocessing, the new block is closed, delimited by '}', after describing "EI( );"

[Effect]

- A disable interrupt function can be created.

[Procedure]

- The #pragma DI and #pragma EI directives are described at the beginning of the C source.
  The following items can be described before #pragma DI and #pragma EI.

  - Comments
  - Other #pragma directives
  - Preprocessing directives that do not generate variable or function definitions or references

- "DI( );" and "EI( );" are written in the source with the same format as a function call.
- DI and EI described after #pragma can be in lowercase or uppercase letters.

[Limits]

- When using this function, DI and EI cannot be used as function names.
  DI and EI are written in uppercase letters. Lowercase letters are treated as ordinary functions. However, if the symbol name case specification option (-CA) is specified during compilation, uppercase and lowercase letters are not distinguished and all are regarded as uppercase letters. Therefore, they are treated as interrupt functions even when described in lowercase letters. For details, see "**CC78K0S Series C Compiler User's Manual Operation**."

[Example]

(C source)

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

(Compiler output object)

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

<When DI and EI are output after and before pre- and postprocessing>

(C source)

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

Interrupt function qualifiers **Interrupt function qualifiers**

(Compiler output object)

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

[Description]

- The main function disables interrupts in the above example.

[Compatibility]

<From another C compiler to this C compiler>

- If interrupt functions are not used, modifications are not needed.
- If interrupt functions are used, the modifications must comform to **Procedure** described above.

<From this C compiler to another C compiler>

- The #pragma DI and #pragma EI directives are deleted or separated by #ifdef. DI and EI can be used as function names. (For example, #ifdef _ _K0S_ _ to #endif)
- If using as an interrupt function, the changes must conform to the compiler specifications.

(13) CPU control instructions

CPU control instructions	CPU control instructions

[Function]

- The following code is output to the object to create the object file.

  (1) Output the HALT instruction.
  (2) Output the STOP instruction.
  (3) Output the NOP instruction.

[Effect]

- The microprocessor's standby function can be used in C programs.
- The clock continues without the CPU operating.

[Procedure]

- The #pragma HALT, #pragma STOP, and #pragma NOP directives are described at the beginning of the C source.
- The following items can be described before the #pragma directives.

  - Comments
  - Other #pragma directives
  - Preprocessing directives that do not generate variable or function definitions or references

- Keywords after #pragma can be described in lowercase and uppercase letters.
- The instructions are described in uppercase letters as shown below in the C source in the same format as a function call.

  (1) HALT( );
  (2) STOP( );
  (3) NOP( );

[Limits]

- When these functions are used, HALT(), STOP(), and NOP() cannot be used as function names.
- HALT, STOP, BRK, and NOP are described in uppercase letters. Functions in lowercase letters are treated as ordinary functions. (However, if the symbol name case specification option (-CA) is specified during compilation, lowercase and uppercase letters are not distinguished and are regarded as uppercase letters. Therefore, even when described in lowercase letters, they are treated as CPU control instructions.)

CPU control instructions **CPU control instructions**

[Example]

(C source)

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

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

(Compiler output object)

```
@@CODE CSEG
_main
       halt
       stop
       nop
```

[Compatibility]

<From another C compiler to this C compiler>

- If CPU control instructions are not used, modifications are not needed.
- To use CPU control instructions, the changes must conform to the **Procedure** described above.

<From this C compiler to another C compiler>

- When the #pragma HALT, #pragma STOP, or #pragma NOP statements are deleted or separated by #ifdef, HALT, STOP, and NOP can be used as function names.
- If using as a CPU control instruction, changes must conform to the compiler specifications.

(14) Absolute address access functions

Absolute address access functions	Absolute address access functions

[Function]

- The code for accessing ordinary RAM space in the object does not call functions. It is expanded inline and output to create the object file.
- If there are no #pragma directives, an absolute address access function is regarded as an ordinary function.

[Effect]

- By using C descriptions, accessing a special address in the ordinary memory space can be simplified.

[Procedure]

- The #pragma access directive is described at the beginning of the C source.
- The format is identical to a function call and is described in the source.
  The four function names for absolute address access are

  peekb, peekw, pokeb, pokew

- The following items can be described before #pragma access.

  - Comments
  - Other #pragma directives
  - Preprocessing directives that do not generate variable or function definitions or references

- The keywords after #pragma can be described in either lowercase or uppercase letters.

[Limits]

- The function name for absolute address access cannot be used as the function name.
- The function for absolute address access is described in lowercase letters. Functions in uppercase letters are treated as ordinary functions. (However, when the symbol name case specification option (-CA) is specified during compilation, lowercase and uppercase letters are not distinguished and are regarded as uppercase letters. Therefore, even when described in uppercase letters, a function is treated as an absolute address access function. For details, see “**CC78K0S Series C Compiler User’s Manual, Operation**.”

Absolute address access functions	Absolute address access functions

[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);
          pokeb (0x1256, 7);
          pokeb (0xfe68, 7);
}
```

(Output assembler source)

```
 :
mov    a,!01234H
mov    !_a,a
mov    a,0FE23H
mov    !_a,a
mov    a,!01256H
xch    a,x
mov    a,!01257H
movw   de,#_b
callt  [@@deist]
movw   ax,0FE68H
callt  [@@deist]
mov    a,#05H ;5
mov    !01234H,a

mov    0FE23H,#05H ;5
mov    a,#07H;7
mov    !01256H,a
mov    0FE68H,#07H ;7
```

Absolute address access functions **Absolute address access functions**

[Absolute address access functions]

(1) unsigned char peekb (addr);
unsigned int addr;

Returns one byte of the addr address.

(2) unsigned int peekw (addr);
unsigned int addr;

Returns two bytes of the addr address.

(3) void pokeb (addr, data);
unsigned int addr;
unsigned char data;

Writes one byte of data in the position pointed to by the addr address.

(4) void pokew (addr, data);
unsigned int addr;
unsigned int data;

Writes two bytes of data in the position pointed to by the addr address.

Note The above function declarations are not affected by the -ZI option.

[Compatibility]

<From another C compiler to this C compiler>

- If absolute address access functions are not used, modifications are not needed.
- To change to absolute address access functions, modifications must conform to **Procedure** described above.

<From this C compiler to another C compiler>

- The #pragma access statements are deleted or separated by #ifdef. The function names for absolute address access can be used as function names.
- If using as an absolute address access function, the changes must conform to the compiler specifications. (#asm, #endasm, or asm( ), etc.)

(15) Bit field declarations

Bit field declarations	**Bit field declarations**

(1) Type specifier extension

[Function]

- A bit field of type unsigned char is not allocated over byte boundaries.
- A bit field of type unsigned int is not allocated over word boundaries. An allocation can cross byte boundaries.
- Bit fields having the same type are allocated in identical byte units (or word units). If their types differ, they are allocated in different word units (or byte units).

[Effect]

- Memory savings, compact object code, and the execution speed is improved.

[Procedure]

- In addition to the unsigned int type, the unsigned char type can be specified as the type specifier for a bit field.

[Compatibility]

<From another C compiler to this C compiler>

- The source does not have to be modified.
- If you wish to use unsigned char in the type specifier, the type specifier is changed.

<From this C compiler to another C compiler>

- If unsigned char is not used in the type specifier, modifications are not needed.
- If unsigned char is used in the type specifier, change it to unsigned int.

(2) Allocation direction for bit fields

[Function]

- The allocation direction of a bit field is set by the -QB option and changes from the most-significant bit (MSB).
- If the -QB option is not specified, allocation is from the least-significant bit (LSB).

[Procedure]

- If the bit field is allocated from the MSB, the -QB option is set during compilation.
- If the bit field is allocated from the LSB, no options are specified.

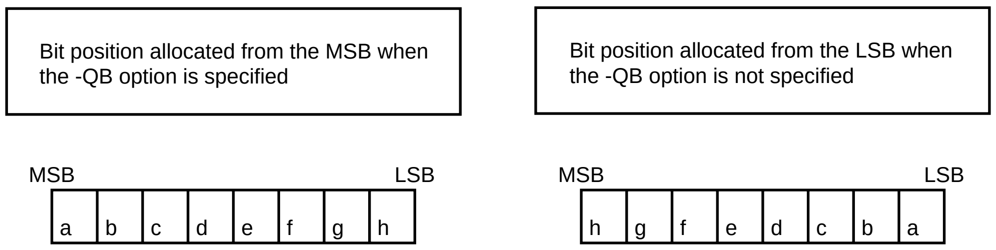
[Example 1]

(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 declarations **Bit field declarations**

Figure 11-1. Bit Position Based on Bit Field Declaration (Example 1)


Bit position allocated from the MSB when the -QB option is specified
Bit position allocated from the LSB when the -QB option is not specified
MSB
LSB
a b c d e f g h
MSB
LSB
h g f e d c b a

[Description]

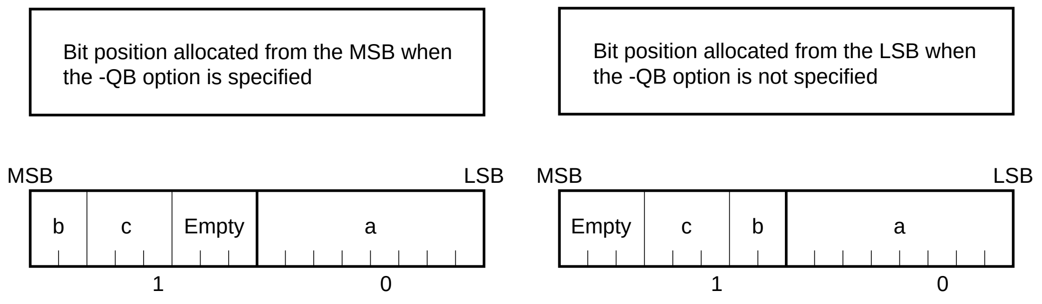
Since a to h are 8 or less bits, they are allocated in a one byte unit.

[Example 2]

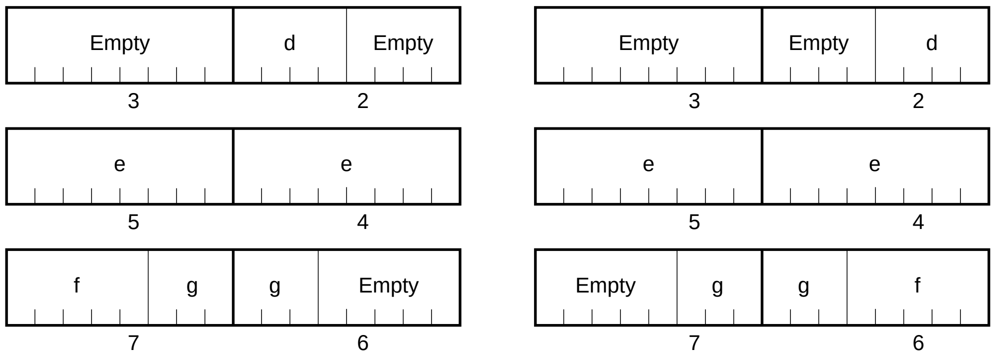
(Bit field declaration)

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

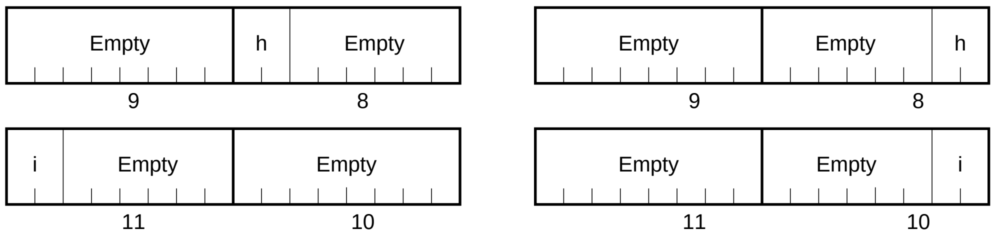
Figure 11-2. Bit Position Based on Bit Field Declaration (Example 2)


Bit position allocated from the MSB when the -QB option is specified
Bit position allocated from the LSB when the -QB option is not specified
MSB
LSB
b
c
Empty
a
1
0
MSB
LSB
Empty
c
b
a
1
0

Member a of type char is allocated at the first byte position. b and c are allocated in the next byte unit. If there is not enough room, the allocation is in the next byte position. Since there are three empty bits and d is four bits, d is allocated to the next byte position.



Since g is a bit field of type unsigned int, byte boundaries can be crossed. 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.

* The numbers below the bit position diagrams indicate the byte offset values from the beginning of the structure.

[Compatibility]

<From another C compiler to this C compiler>

- Modifications are not needed.

<From this C compiler to another C compiler>

- If -QB option is specified and coding considers the order in which bit fields are allocated, changes are necessary.

(16) Changing the compiler output section name

#pragma section... **#pragma section...**

[Function]

- The compiler output section name is changed, and the starting address is set. If the starting address is omitted, the default repositioning attribute is enabled.
- If the @@CALT section name is changed with the AT starting address setting, the callt function must be described before or after other functions in the source file.

[Effect]

- By changing the compiler output section name, that section can be independently positioned.

[Procedure]

- The section name changed by the following #pragma directive, the section name after the change, and the starting address of the section are set.
  This #pragma directive is described at the beginning of the C source file.
  If the #pragma PC(type) is described, #pragma directives are described later. The following items can be described before this #pragma directive.

  - Comments
  - Preprocessing directives that do not generate variable or function definitions or references

  This declaration is placed at the beginning of the file.

```
#pragma section compiler-output-section-name changed-section-name [AT-starting-address]
```

- Always describe the compiler-output-section-name in uppercase letters in the keywords after #pragma. The section and AT can be described in lowercase or uppercase letters.
- The format for the changed-section-name conforms to the assembler specifications.
- Only hexadecimal C language integer constants and assembler numerical constants can be described in the starting address.

#pragma section... **#pragma section...**

[C language integer constants]

```
<integer-constant> ::= <hexadecimal-number>
<hexadecimal-number> ::=  0xn
                        | 0xn ... n
                        | 0Xn
                        | 0Xn ... n
                        (n=0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
```

[Assembler numerical constants]

```
<integer-constant> ::= <hexadecimal-number>
<hexadecimal-number> ::=  nH
                        | n ... nH
                        | nh
                        | n ... nh
                        (n=0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E,F)
```

- The first hexadecimal character must be a number.

(Example) When the value of 255 is expressed in hexadecimal, zero must be specified before F to become 0FFH.

[Example]

(C source)

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

void main()
{
         Function body
}
```

(Output object)

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

#pragma section... **#pragma section...**

[Description]

The @@CODE section name is changed to CC1. The starting address is set to address 2400H.

[Compatibility]

<From another C compiler to this C compiler>

- If the change section name function is not supported, modifications are not needed.
- To change the section name, the modifications must conform to **Procedure** described above.

<From this C compiler to another C compiler>

- #pragma section... is deleted or separated by #ifdef.
- To change the section name, the change must conform to the specifications of each compiler.

[Limits]

- A section name indicating the vector table segment (for example, @@VECT02) cannot be changed.
  A section name indicating the bank function segment
  (for example, @@BANK1) cannot be changed.
- Multiple versions of the same section name as the AT starting address specification (for example, in other files) generate link errors.
- The compiler restricts the address range to the starting address specification for the @@DATS, @@BITS, and @@INIS changed section names.

  (Address range)
  0xfe20 - 0xfeB7

(Warnings)

- The section corresponds to the segment in the assembler.
- The compiler does not check for duplications of the changed section names and other symbols. Consequently, the user assembles the output assembler list and should verify that there are no duplicates.
- If the ROM-related section name (*) is changed by using #pragma section when programming the ROM, the user is responsible for changing the start-up routine.

#pragma section... **#pragma section...**

(*) ROM-related section names

```
@@R_INIT, @@R_INIS, @@INIT, @@INIS
```

- If there is no setting in the ROM because of the -nr option, @@R_INIT and @@R_INIS are not generated.
- Examples of changing the start-up routine and termination module by changing the ROM-related section are explained next.

[Example of Changing the Start-Up Routine When Changing the ROM-related Section Name]

Examples of changing the start-up routine (cstartr.asm or cstartrn.asm) and the termination module (rom.asm) when changing ROM-related section names are illustrated.

(C source)

```
#pragma section @@ R_INIT RTT1
#pragma section @@ INIT TT1
```

When the above #pragma section description changes the section name where the external variables are initialized and stored, the user must add the initialization for the external variables stored in the changed section to the start-up routine.
The start-up routine includes a section for copying the declarations for the starting label of the changed section and the initial values, and a section for declaring the ending label in the termination module. The following shows its procedure
RTT1_S and RTT1_E are the starting and ending labels of the RTT1 section. TT1_S and TT1_E are the starting and ending labels of the TT1 section.

#pragma section... **#pragma section...**

(Changing point of the cstartx.asm start-up routine)

(1) The declaration of the ending label of the section with the changed name is added.

```
                    :
      EXTRN _main, _@STBEG, _hdwinit
$_IF (EXITSW)
        EXTRN _exit
$ENDIF
                    :
        EXTRN _?R_INIT, _?R_INIS, _?DATA, _?DATS
        EXTRN RTT1_E, TT1_E    ← Add the EXTRN declurations for RTT1_E and TT1_E.
                    :
```

(2) A section is added to copy the initial values from the RTT1 section with the changed name to the TT1 section.

```
LDATS1:
      CMPW  HL, DE
      BE    $LDATS2
      MOV   [DE+], A
      BR    $LDATS1
LDATS2:
      MOVW  DE, #TT1_S
      MOVW  HL, #RTT1_S
LTT1:
      CMPW  HL, #RTT1_E
      BE    $LTT2
      MOV   A, [HL+]
      MOV   [DE+], A
      BR    $LTT1
LTT2:
      CALL  !_main              ;main();
        :
```

Added section to copy the initial values from the RTT1 section to the TT1 section.

#pragma section... **#pragma section...**

(3) Sets the starting label of the section with the changed name.

```
    .
    .
    .
@@R_INIT     CSEG
_@R_INIT:
@@R_INIS     CSEG  UNITP
_@R_INIS:
@@INIT       DSEG
_@INIT:
@@DATA       DSEG
_@DATA:
@@INIS       DSEG  SADDRP
_@INIS:
@@DATS       DSEG  SADDRP
_@DATS:

RTT1         CSEG
RTT1_S:      Add the label setting indicating the beginning of the RTT1 section.

TT1          DSEG
TT1_S:       Add the label setting indicating the beginning of the TT1 section.

@@CODE       CSEG
@@CALT       CSEG  CALLT0
@@CNST       CSEG
@@BITS       BSEG
;
END
```

#pragma section... **#pragma section...**

(Changing point of the rom.asm termination module)

(1) 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               ← Add RTT1_E and TT1_E.

;
@@R_INIT     CSEG
-?R_INIT:
@@R_INIS     CSEG  UNITP
_?R_INIS:
@@INIT       DSEG
_?INIT:
@@DATA       DSEG
_?DATA:
@@INIS       DSEG  SADDRP
_?INIS:
@@DATS       DSEG  SADDRP
_?DATS:
             :
```

(2) Setting the label indicating the end

```
           :
RTT1   CSEG
RTT1_E:       Adds the label setting indicating the end of the RTT1 section.

TT1    DSEG
TT1_E:        Adds the label setting indicating the end of the TT1 section.
 ;
       END
```

Next, all sections output by the compiler are illustrated.

Table 11-12. Compiler Output Section Names

Section name	Segment type	Repositioning attributes	Description
@@CODE	CSEG		Code segment
@@CNST	CSEG		const variable segment
@@R_INIT	CSEG		Initialization data segment (initial values)
@@R_INIS	CSEG	UNITP	Initialization data segment (initial values, sreg variables)
@@CALT	CSEG	CALLT0	Segment for callf function table
@@VECT06	CSEG	AT 0006H	Vector table segment
@@INIT	DSEG		Temporary data segment (initial values)
@@DATA	DSEG		Temporary data segment (no initial values)
@@INIS	DSEG	SADDRP	Temporary data area segment (initial values, sreg variables)
@@DATS	DSEG	SADDRP	Temporary data area segment (no initial values, sreg variables)
@@BITS	BSEG		boolean type variable and bit type variable segments

* When specified in a ROM, @@R_INIT and @@R_INIS are output.
* @@VECT06 is an example of specifying the interrupt caused by a NMI due to #pragma vect(interrupt).

Each segment type is allocated as shown below if there is no repositioning attribute.

Table 11-13. Segment Placement Destination

Segment Type	Placement Destination
CSEG	ROM
BSEG	RAM saddr area
DSEG	RAM

(17) Binary constants

Binary constants	**Binary constants 0bxxx**

[Function]

- A binary representation is added to the syntax for integer constants.
- A binary number can be described in any position where an integer constant can be described.

[Effect]

- If you wish to describe constants in a bit array, the readability improves when direct descriptions are possible without converting to octal or hexadecimal numbers.

[Procedure]

- Binary numbers are described in the C source. Binary constants are described as follows:

```
0b [binary-number]
0B [binary-number]
```

***** Binary number: Either 0 or 1

- A binary constant is 0b or 0B followed by the numbers 0 or 1.
- The value of a binary constant is computed basing on 2.
- The type of the binary constant is the first type in the following list that can represent the value.

• Binary number without subscript	:	int, unsigned int, long int, unsigned long int
• u or U subscript	:	unsigned int, unsigned long int
• l or L subscript	:	long int, unsigned long int
• u or U subscript and l or L subscript	:	unsigned long int

[Example]

(C source)

```
   unsigned     i;
   i = 0b11100101;
The compiler output object is the same as below.
   unsigned     i;
   i = 0xE5;
```

Binary constants **Binary constants 0bxxx**

[Compatibility]

<From another C compiler to this C compiler>

- Modifications are not needed.

<From this C compiler to another C compiler>

- If the compiler supports binary constants, the changes must conform to the compiler specifications.
- If the compiler does not support binary constants, modifications must conform to the other integral formats of octal, decimal, and hexadecimal.

(18) Change module name [Function]

Change module name function	#pragma name

[Function]

- The first eight characters of the module name specified in the symbol information table of the object module file is output to the symbol table entry.
- When -g is specified in the assembler list file, the first eight characters of the specified module name are output as the symbol information (MOD_NAM). When -ng is specified, they are output as the NAME pseudo-instruction.
- If a module name that is nine characters or longer, a warning message is output.
- If illegal characters are described, an error occurs and processing aborts.
- If this #pragma directive occurs more than once in one source file, a warning message is output and the last description becomes valid.

[Effect]

- The module name of the object can be changed to any name.

[Procedure]

- The description is

  #pragma name module-name

  The module name consists of the characters that are allowed for file names in the operating system, excluding the parentheses () and kanji. Lowercase and uppercase letters are not distinguished, but all letters are regarded as uppercase letters when the -CA option is specified.

[Example]

```
#pragma name module1
         :
```

[Limits]

- If kanji is included in the first eight characters of the input file name, an error occurs and processing aborts unless the module name is changed by this #pragma directive.

[Compatibility]

<From another C compiler to this C compiler>

- If the change module name function is not supported, modifications are not needed.
- To change the module name, the modification must conform to **Procedure** described above.

<From this C compiler to another C compiler>

- #pragma name... is deleted or separated by #ifdef.
- If the module name is changed, the changes must conform to the compiler specifications.

(19) Rotate functions

Rotate functions **#pragma rot**

[Function]

- Code that rotates the value of an expression in an object is not a function call, but is directly expanded inline and output to create the object file.
- If there are no #pragma directives, the rotate function is considered to be an ordinary function.

[Effect]

- Even if processing for rotating is not specified by C source or ASM descriptions, the rotate function can be implemented.

[Procedure]

- The function is described in the source in the same format as a function call. The four rotate function names are

  rorb, rolb, rorw, rolw

  (Details about rotate functions are described later.)
- The use of a rotate function is declared by the #pragma rot directive in the module. However, the following items can be described before #pragma rot.
  - Comments
  - Other #pragma directives
  - Preprocessing directives that do not generate variable or function definitions or references
- The keywords after #pragma can be either lowercase or uppercase letters.

[Example]

(C source)

```
#pragma rot
unsigned char a = 0x11;
unsigned char b = 2;
unsigned char c;
vold 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
```

Rotate functions **#pragma rot**

[Rotate functions]

(1) unsigned char rorb (x, y);
unsigned char x;
unsigned char y;

x is rotated to the right y times.

(2) unsigned char rolb (x, y);
unsigned char x;
unsigned char y;

x is rotated to the left y times.

(3) unsigned int rorw (x, y);
unsigned int x;
unsigned char y;

x is rotated to the right y times.

(4) unsigned int rolw (x, y);
unsigned int x;
unsigned char y;

x is rotated to the left y times.

Note The above function declarations are not affected by the -ZI option.

[Limits]

- The rotate function name cannot be used as a function name.
- The rotate function is described in lowercase letters. Functions in uppercase letters are treated as ordinary functions. (However, if the symbol name case specification option (-CA) is specified during compilation, lowercase and uppercase letters are not distinguished and are regarded as uppercase letters. Therefore, functions in uppercase letters are treated as rotate functions.)

Rotate functions **#pragma rot**

[Compatibility]

<From another C compiler to this C compiler>

- If the rotate functions are not used, modifications are not needed.
- To change to the rotate functions, the modifications must conform to **Procedure** described above.

<From this C compiler to another C compiler>

- The #pragma rot statements are deleted or separated by #ifdef. A rotate function name can be used as a function name.
- If using as a rotate function, the modifications must conform to the compiler specifications. (#asm, #endasm, or asm( ), etc.)

(20) Multiplication Function

Multiplication function **#pragma mul**

[Function]

- Outputs a code calling a library which multiplies the value of an expression in an object.
- If there are no #pragma directives, the multiplication function is regarded as an ordinary function.

[Effect]

- Since code that produces the data size of I/O for the multiplication directive is generated, the execution speed can be improved and compact code can be generated than with an ordinary multiplication expression.

[Procedure]

- The same format as in a function call is described in the source. The multiplication function name is mulu.

```
unsigned int mulu (x, y);
unsigned char x;
unsigned char y;
```

x and y are multiplied without signs.

- The use of the multiplication function is declared by the #pragma mul directive in the module. However, the following items can be described before #pragma mul.

    - Comments
    - Other #pragma directives
    - Preprocessing directives that do not generate variable or function definitions or references

- The keywords described after #pragma can be lowercase or uppercase letters.

[Limits]

- CC78K0 expands a multiplication code in inline. However, this compiler does not perform an inline expansion, but call a library. The description is as same as CC78K0.
- Multiplication function names cannot be used as function names (when #pragma mul is declared).
- A multiplication function is described in lowercase letters. Functions in uppercase letters are treated as ordinary functions. (However, if the symbol name case specification option (-CA) is specified during compilation, lowercase and uppercase letters are not distinguished and are regarded as uppercase letters. Therefore, functions in uppercase letters are treated as multiplication functions.)

Multiplication functions **#pragma mul**

[Example]

(C source)

```
#pragma mul
unsigned char a = 0x11;
unsigned char b = 2;
unsigned int i;
void main()
{
         i = mulu (a, b);
}
```

(Output assembler source)

```
mov    a, !_b
xch    a, x
mov    a, !_a
callt  [@@ mulu]
movw   hl, #_i
callt  [@@ hlist]
```

[Compatibility]

<From another C compiler to this C compiler>

- If multiplication functions are not used, modifications are not needed.
- To change to the multiplication function, the changes must follow the description method described above.

<From this C compiler to another C compiler>

- The #pragma mul statements are deleted or separated by #ifdef. A multiplication function name can be used as a function name.
- If using as a multiplication function, the modifications must conform to the compiler specifications. (#asm, #endasm, or asm(), etc.)

(21) Division Function

Division function	#pragma div

[Function]

- Outputs a code calling a library which devides the value of an expression in an object.
- If there are no #pragma directives, the division function is regarded as an ordinary function.

[Effect]

- Since code that produces the data size of the I/O for the division directive is generated, the execution speed can be improved and compact code can be generated than in an ordinary division expression.

[Procedure]

- The same format as a function call is described in the source. The two division function names are divuw and moduw.

[Division Functions]

```
(1)  unsigned int divuw (x, y);
     unsigned int x;
     unsigned char y;

     x and y are divided without signs and the quotient is returned.

(2)  unsigned char moduw (x, y);
     unsigned int x;
     unsigned char y;

     x and y are divided without signs and the modulus is returned.
```

- The use of the division function is declared by the #pragma div directive in the module. However, the following items can be described before #pragma div.

  - Comments
  - Other #pragma directives
  - Preprocessing directives that do not generate variable or function definitions or references

- The keywords described after #pragma can be lowercase or uppercase letters.

Division function **#pragma div**

[Limits]

- CC78K0 expands a division code in inline. However, this compiler does not perform an inline expansion, but call a library. The description is as same as CC78K0.
- Division function names cannot be used as function names.
- The division function is described in lowercase letters. Functions in uppercase letters are treated as ordinary functions. (However, if the symbol name case specification option is specified during compilation, lowercase and uppercase letters are not distinguished and are regarded as uppercase letters. Therefore, uppercase letters are treated as division functions. See the "**CC78K0 User's Manual, Operation**.")

[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 assembler source)
```
     mov    a, !_b
     mov    c, a
     movw   hl, #_a
     callt  [@@ hlilo]
     callt  [@@ divuw]
     movw   hl, #_i
     callt  [@@ hlist]
     mov    a, !_b
     movw   hl, #_a
     callt  [@@ hlilo]
     callt  [@@ divuw]
     mov    a, c
     mov    !_c, a
```

Division function **#pragma div**

[Compatibility]

<From another C compiler to this C compiler>

- If division functions are not used, modifications are not needed.
- To change to division functions, the modifications must conform to the description method described above.

<From this C compiler to another C compiler>

- The #pragma div statements are deleted or separated by #ifdef. A division function name can be used as a function name.
- If using as a division function, the modifications must conform to the specifications of each compiler (#asm, #endasm, or asm( ), etc.).

(22) Data insertion [Function]

Data insertion function **#pragma opc**

[Function]

- Constant data is inserted at the current address.
- If there are no #pragma directives, data insertion functions are regarded as ordinary functions.

[Effect]

- Even when asm descriptions are not used, special data or instructions can be inserted in the code area. If the asm description is used, the object cannot be obtained without passing through the assembler. However, when the data insertion function is used, the object can be obtained even without passing through the assembler.

[Procedure]

- The same format as a function call is described in the source. The data insertion function name is __OPC. (For details on data insertion functions, see “Data Insertion Functions” to be described later.)
- The use of data insertion functions is declared by the #pragma opc directive. However, the following items can be described before #pragma opc.

  - Comments
  - Other #pragma directives
  - Preprocessing directives that do not generate variable or function definitions or references

- The keywords after #pragma can be lowercase or uppercase letters.

[Limits]

- A data insertion function name cannot be used as a function name (when #pragma opc is specified).
- _ _OPC is described in uppercase letters. Functions in lowercase letters are treated as ordinary functions. (However, if the symbol name case specification option (-CA) is specified during compilation, lowercase and uppercase letters are not distinguished and are regarded as uppercase letters. Therefore, functions in lowercase letters are treated as data insertion functions.)

Data insertion function #pragma opc

[Example]

(C source)

```
#pragma opc
void main()
{
         _ _OPC (0xBF);
         _ _OPC (0xA1, 0x12);
         _ _OPC (0x10, 0x34, 0x12);
}
```

(Output assembler source)

```
_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
```

[Data Insertion Functions]

(1) `void_ _OPC (unsigned char x, ...);`

The constant value described in the argument is inserted at the current address.
Arguments can only be specified as constants.

[Compatibility]

<From another C compiler to this C compiler>

- If data insertion functions are not used, modifications are not needed.
- To change to data insertion functions, the modifications must conform to **Procedure** described above.

<From this C compiler to another C compiler>

- The #pragma opc statements are deleted or separated by #ifdef. A data insertion function name can be used as a function name.
- If using as a data insertion function, the modifications must conform to the specifications of each compiler (#asm, #endasm, or asm( ), etc.).

(23) Static Model

Static Model	**Static Model**

[Function]

- All arguments are passed to registers. (See **Appendix C "FUNCTION CALL INTERFACE"**)
- Function arguments passed via registers are allocated to function-specific static areas.
- Automatic variables are allocated to function-specific static areas.
- In the case of a leaf functionNote 1, arguments and automatic variables are allocated from the high-order address to the saddr area below and including 0FEFFH.
  This saddr area is shared by the leaf function of all modules, and therefore it is called a common area.
  The maximum size of a common area can be specified with a parameter when specifying the -SM option.

```
-SM[nn]  (nn:0 to 16)
```

  nn bytes are allocated as the common area, and the remaining bytes are allocated to function-specific static areas. If nn = 0 or is omitted, there is no common area.
  The sreg/_ _sreg keyword can be added to function arguments and automatic variables. Function arguments and automatic variables to which the sreg/_ _sreg keyword has been added are allocated to the an saddr area and can be bit-manipulated.
- Function arguments and automatic variables (excluding static variables inside functions) can be allocated to saddr and bit manipulated.
- The following macro definitions are automatically performed by the compiler.
  #define _ _STATIC_MODEL_ _ 1

Note 1. Function that does not call a function. The compiler performs judgment automatically, therefore it is not necessary to describe norec/_ _leaf keywords. (When static model is specified).

[Effect]

- Normally, instructions to access static areas are shorter and faster than instructions to access stack frames, and therefore enable the reduction of object code as well as faster execution.
- Saving and restoring of argument and variables (interrupt function register variables, norec function arguments/automatic variables, run-time library arguments) done in the normal model is not performed in the static model, which allows faster interrupt processing.
- Plural leaf functions share the same data area, thereby saving memory space.

Static Model **Static Model**

[Procedure]

- During compiling, specify the -SM option. Objects at this time are called static model objects, whereas objects when the -SM is not specified during compiling are called normal model objects.

[Limits]

- Static model modules cannot be linked to normal model modules. However, static model modules can be linked to each other even if their common areas are of different sizes.
- Floating-point numbers are not supported. If the float and double keywords are described, a fatal error results.
- Up to 3 arguments, or a total of 6 bytes, can be passed.
- Arguments are not passed via stacks, therefore the variable length argument "..." cannot be used. Variable length arguments result in errors.
- Structure and union arguments and return values cannot be used. Describing them results in errors.
- The noauto, norec and _ _leaf functions cannot be used. Their description causes a warning to be output, and they are ignored.
- The recursive function cannot be used. Function arguments and automatic variable areas are reserved statically, therefore the recursive function cannot be used.
- Prototype declarations cannot be omitted. Regardless of the presence of a function call, an error results in the absence of a definition of the function and a prototype declaration.
- Functions that are recursive functions, and limits on arguments and return values, cannot be used, and therefore one part of the standard library cannot be used.

Static Model **Static Model**

[Example]

(C source)

```
[when specifying the -SM option]
   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=p1;
            a2=p2+p3;
   }
```

(Compiler output object)

```
@@DATA        DSEG
?L0003: DS    (1)                      ; Automatic variable i of main function
?L0004: DS    (1)                      ; Automatic variable j of main function
?L0005: DS    (1)                      ; Automatic variable k of main function
?L0008: DS    (1)                      ; Automatic variable a2 of sub function
               :
; line  1:    void sub (char, char, char);
; line  2:    void main(){

@@CODE        CSEG
_main:
; line  3:          char i=1;
       mov    a,#01H ;1
       mov    !?L0003,a    ;i          ; Automatic variable I
; line  4:          char j,k;
```

Static Model **Static Model**

(Compiler output object)

```
; line   5:           j=2;
        inc    a
        mov    !?L0004,a     ;j         ; Automatic variable j
; line   6:           k=i+j;
        add    a,!?L0003     ;l         ; Add i and j
        mov    !?L0005,a     ;k         ; Assign to k
; line   7:           sub(i,j,k);
        movw   hl,ax                    ; Pass k by register h
        mov    a,!?L0004     ;j
        movw   bc,ax                    ; Pass j by register b
        mov    a,!?L0003     ;l         ; Pass I by register a
        call   !_sub
; line   8:    }
        ret
; line   9:
; line  10:    void sub (char p1, char p2, char p3){
_sub:
        mov    _@KREG15,a               ; Allocate 1st argument to common area
        movw   ax,bc
        mov    _@KREG14,a               ; Allocate 2nd argument to common area
        movw   ax,hl
        mov    _@KREG13,a               ; Allocate 3rd argument to common area
; line  11:           char a1,a2;
; line  12:           a1=p1;
        mov    a,_@KREG15    ;p1        ; 1st argument _@KREG15
        mov    _@KREG12,a    ;a1        ; Automatic variable a1

; line  13:           a2=p2+p3;
        mov    a,_@KREG14    ;p2        ; Automatic variable p2
        add    a,_@KREG13    ;p3        ; Add automatic variable p3
        mov    !?L0008,a     ;a2        ; Automatic variable a2 is a function-
                                          specific area
; line  14:    }
        ret
```

Static Model **Static Model**

[Compatibility]

<From another C compiler to this C compiler>

- If creating normal model objects, modifications of the source are not required if the -SM option is not specified.
- If creating static model objects, follow the Procedure above.

<From this C compiler to another C compiler>

- If simply compiling on another compiler, modifications of the source are not required.

(Warnings)

<1> Arguments and automatic variables are reserved statically, therefore the recursive function may cause the contents of arguments and automatic variables to be destroyed. It should be kept in mind that a function directly calling itself causes an error, but that in the case of a function calling itself from another function it called, the compiler cannot detect this and thus no error is caused.

<2> Cautions on case when function call from interrupt function occurs
If a function call from an interrupt function occurs, attention must be paid to the following.

- If functions that are being processed when an interrupt is issued are called by interrupt processing (interrupt function and function called by interrupt function), the argument and automatic variable may be destroyed.
- Even when functions that are being processed when an interrupt is issued use a common area, saving and restoring of the common area is not performed, therefore care is required when using a common area in interrupt processing (function called by interrupt function).
  If possible, describing by performing only flag setting without performing function call from the interrupt function, and referencing that flag in the main function, is recommended.

(24) Changing type

Static Model **Static Model**

(1) Changing int and short type to char type

[Function]

- int/short types are treated as char type.
- Details on type changes are shown below. (Some type changes are influenced by the -QU option)

Table 11-14. Type Changes Using -ZI and -QU Options

Type described in C source	Option	Type after change
short, short int, int	-QU present	unsigned char
short, short int, int	-QU absent	signed char
unsigned short, unsigned short int, unsigned, unsigned int		unsigned char
signed short, signed short int signed, signed int		signed char

- A warning is output the first time an int or short keyword appears in the C source.
- The -QC option is enabled regardless of whether it is specified or not. If the -QC option is not specified, a warning is output and the -QC option becomes enabled.
- If -ZAI, etc., is specified at the same time as the -ZA option, a warning is output. (only when -W2 is specified)
- The elements listed below, which have a syntax that can be described by a type specifier and be omitted, are treated as char type.
  Function arguments and return values
  Variables and function declarators with type specifier omitted
- The following macro definitions are automatically performed by the compiler.
  #define _ _FROM_INT_TO_CHAR_ _ 1
- One part of the standard library cannot be used.

Static Model	**Static Model**

[Procedure]

- Specify the -ZI option.

[Limits]

- Modules for which the -ZI option is specified and modules for which the -ZI is not specified cannot be linked.

Static Model **Static Model**

(2) Changing long type to int type

[Function]

- long type is treated as int type.
- Details on type changes are shown below.

Table 11-15. Type Changes Using -ZL Option

Type described in C source	Type after change
unsigned long, unsigned long int	unsigned int
long, long int signed long, signed long int	signed int

- A warning is output the first time a long keyword appears in the C source.
- A warning is output if -ZAL, etc., is specified at the same time as the -ZA option.
- The following macro definitions are automatically performed by the compiler.
  #define _ _FROM_LONG_TO_INT_ _ 1
- One part of the standard library cannot be used.

[Procedure]

- Specify the -ZL option.

[Limits]

- Modules for which the -ZL option is specified and modules for which the -ZL option is not specified cannot be linked.

11.5 Modifying the C Source

By using extended functions, efficient object code can be created. However, since extended functions are adapted to the 78K/0S Series, other uses necessitate modifications. Porting from another C compiler to this C compiler, or from this C compiler to another C compiler is described.

<From another C compiler to this C compiler>

- #pragma
  If another C compiler supports #pragma, the C source must be modified. The modification is examined based on the specifications of the C compiler.
- Extended specifications
  If the specifications are extended, such as the addition of keywords by the other C compiler, modifications are required. The modification is examined based on the specifications of the C compiler.

<From this C compiler to another C compiler>

Since this C compiler adds keywords as extended functions to port to another C compiler, the keywords are deleted or are separated by #ifdef.

[Examples]

(1) Disable keywords (same for callf, sreg, noauto, norec, etc.)

```
#ifndef _ _K0S_ _
        #define callt       /* Make callt function an ordinary function. */
#endif
```

(2) Change to another type.

```
#ifndef _ _K0S_ _
        #define bit char     /* Change bit type variable to char type variable. */
#endif
```

[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 RA78K0S Series 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) Warnings

12.1 Accessing Arguments/Automatic Variables

The procedure to access arguments and automatic variables of this C compiler is described below.

12.1.1 Normal model

- On the function call side, register arguments are passed together with regular arguments in the same way. The first argument uses the following registers and stacks, and subsequent arguments are passed via stacks.

Table 12-1. Passing Variables (Function Call Side)

Type	Passing Location (First Argument)	Passing Location (Second and Later Arguments)
1-byte, 2-byte data	AX	Stack passing
4-byte data	AX, BC	Stack passing
Floating-point number	AX, BC	Stack passing
Others	Stack passing	Stack passing

***** 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.
  Arguments passed via a regular register are pushed to a stack. 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 to the following registers, saddr areas, or stack frames using an option. The base pointer used when storing to a stack frame uses the HL register.
  If the function argument is register declared 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)	Declared argument or automatic variable (including register variables)	Argument: _@KREG12 to 15 [0FEE4H to 0FEE7H] auto variable: _@KREG00 to 11 [0FED8H to 0FEE3H], _@KREG12 to 15 not allocated to argument	Only number of bytes of variable or argument is allocated, in order of appearance
-QRV	Declared argument or automatic variable (including register variables)	HL register, Argument: _@KREG12 to 15 [0FEE4H to 0FEE7H] auto variable: _@KREG00 to 11 [0FED8H to 0FEE3H] _@KREG12 to 15 not allocated to argument	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      _@KREG10
    EXTRN      _@KREG14
    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 calling
    pop        ax                         ; Argument passed on stack
    ret

_func0:
    push       hl
    xch        a,x
    xch        a,_@KREG12
    xch        a,x
    xch        a,_@KREG13                 ; Allocate register argument to _@KREG12.
    push       ax                         ; Save the saddr area for register
                                            arguments.
    movw       ax,_@KREG10
    push       ax                         ; Save the saddr area for automatic
                                            variables.
    movw       ax,_@KREG14
    push       ax                         ; Save the saddr area for register
                                            variables.
    movw       ax,sp
    movw       hl,ax
    mov        a,[hl+10]                  ; Argument p2 passec on stack
    xch        a,x
    mov        a,[hl+11]
    movw       hl,ax
    movw       ax,hl
    movw       _@KREG14,ax        ;r      ; Assigned to register variables.
    movw       ax,_@KREG12        ;p1     ; Register argument p1
    movw       _@KREG10,ax        ;a      ; Assigned to automatic variable a.
    pop        ax
    movw       _@KREG14,ax                ; Restore the saddr area for register
                                            variables.
    pop        ax
    movw       _@KREG10,ax                ; Restore the saddr area fo automatic
                                            variables.
    pop        ax
    movw       _@KREG12,ax                ; Restore the saddr area for register
                                            arguments.
    pop        hl
    ret
    END
```

12.1.2 Static model

- On the function call side, register arguments are passed together with regular arguments in the same way.
- 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 HL		

***** 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	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 last one, automatic variables are allocated by order of appearance
Default	Argument, register variable declared with register	DE register	Only number of bytes of variable or argument is allocated, in order of appearance

The following example shows the function call.

(C source: Static Model at -SM and -QV specifications)

```
   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;
   }
```

(Output assember source)

```
   PUBLIC _func
   PUBLIC _main
              :
@@DATA        DSEG
?L0005:       DS   (1)                 ; Argument p2
?L0006:       DS   (1)                 ; Automatic 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 to de.
   movw       ax,bc
   mov        !?L0005,a                ; Copy argument p2 to ?L0005.
   mov        a,!?L0005        ;p2     ; Assigned to automatci variable a
   mov        !?L0006,a        ;r      ; Restore register for register arguments.
   movw       ax,de                    ;
   mov        !?L0007+1,a      ;a
   xch        a,x
   mov        !?L0007,a        ;a      ;
   pop        de                       ;
   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, 2-byte integer	BC	AX
4-byte integer	BC (low-order), DE (high-order)	AX (low-order), BC (high-order)Note 1
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

Note 1. Not supported in V1.00.

12.3 Calling Assembly Language Routines from the C Language

This section shows examples when the normal model (default) is used. It should be remembered that, if the -QV option, -QR option, and -QRV option are specified, arguments are stored as indicated in Table 12-1. 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.

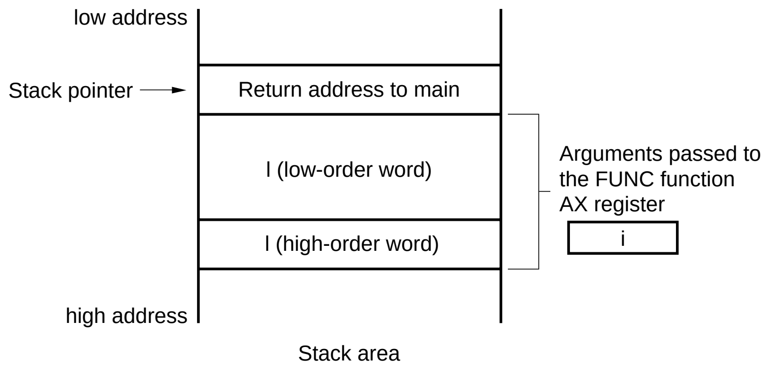
```
extern int FUNC(int, long);     /* Function prototype */

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 arguments passed from the main function to the FUNC function 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.
(2) Save the work register.
(3) Copy the stack pointer (SP) to the base pointer (HL).
(4) Perform the processing in the FUNC function.
(5) Set the return value.
(6) Restore the saved register.
(7) Return to the main function.

Next, an example of an assembly language program is explained.

```
$PROCESSOR(9024)

    PUBLIC _FUNC
    PUBLIC _DT1
    PUBLIC _DT2

@@DATA        DSEG
?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
    MOV       !_DT1,A                   ;move 1st argument(i)
    XCH       A,X
    MOV       A,[HL+1]                  ;arg1
    MOV       !_DT1+1,A
    MOV       A,[HL+8]                  ;arg2
    XCH       A,X
    MOV       A,[HL+9]                  ;arg2
    MOVW      BC,AX
    MOV       A,[HL+6]                  ;arg2
    XCH       A,X
    MOV       A,[HL+7]                  ;arg2
    MOVW      DE,#_DT2
    XCH       A,X
    MOV       [DE],A                    ;move 2nd argument(l)
    XCH       A,X
    INCW      DE
    MOV       [DE],A
    XCHW      AX,BC
    INCW      DE
    XCH       A,X
    MOV       [DE],A
    XCH       A,X
    INCW      DE
    MOV       [DE],A
    XCHW      AX,BC
    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. During assembling, if an option (-nca) distinguishing between uppercase and lowercase characters is specified, descriptions do not need to be made with uppercase characters. 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)

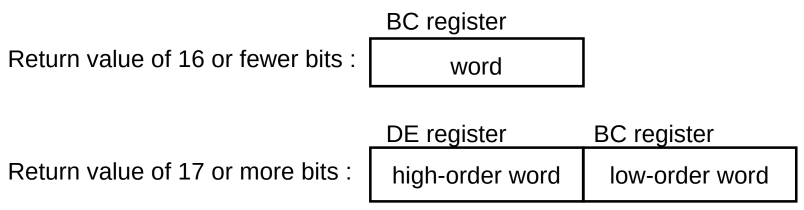
The stack pointer (SP) changes due to pushing and popping inside functions. Therefore, the stack pointer is copied to register HL and used as the base pointer of arguments.

(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.

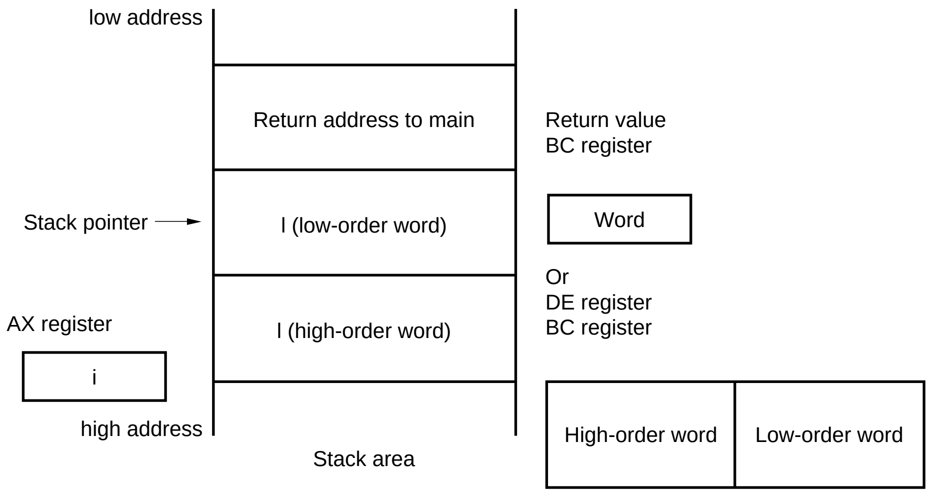

Return value of 16 or fewer bits :
BC register
word
Return value of 17 or more bits :
DE register
high-order word
BC register
low-order word

(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).

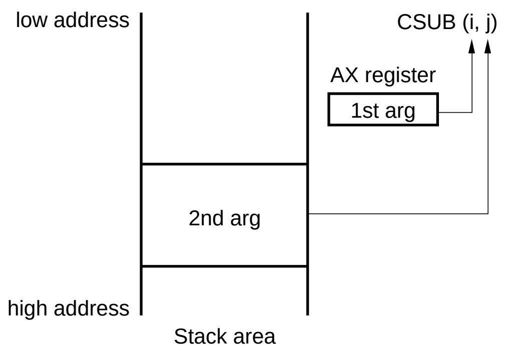
This is an example of an assembly language program.

```
$PROCESSOR (9024)

      NAME   FUNC2
      EXTRN  _CSUB
      PUBLIC _FUNC2

@@CODE CSEG
_FUNC2:
      movw   ax, #20H             ; seg 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 the Stack



(2) Saving the work registers (AX, BC, and DE)

The four 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

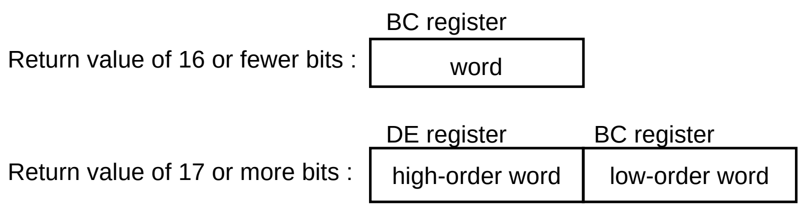
A CALL instruction calls a C language function. If the C language function is a callt function, the callt instruction performs the call.

(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.

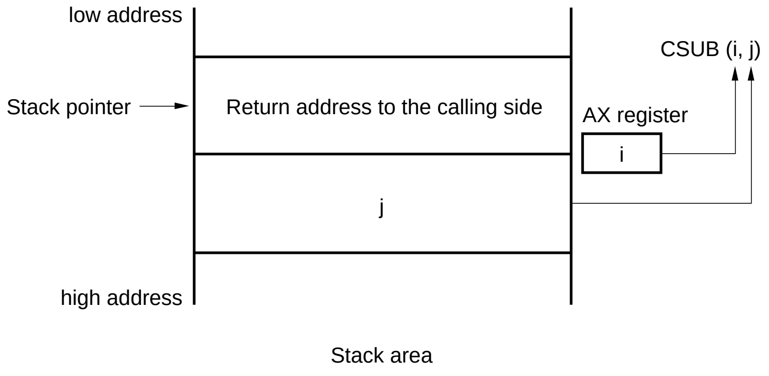

BC register
Return value of 16 or fewer bits :
word
DE register
BC register
Return value of 17 or more bits :
high-order word
low-order word

(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, “Passing Arguments to the C Language.”

```
void CSUB (i. j)
int     i, j ;
{
        i += j;
}
```

Figure 12-4. Passing Arguments to the 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 extrn 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 RA78K0S assembler. (In this example, the -nca option that differentiates lowercase and uppercase letters must be specified during assembly. -nca is the default.)

```
$PROCESSOR (9024)

     PUBLIC _subf
     EXTRN  _c
     EXTRN  _i

@@CODE CSEG
_subf:
     MOV    a, #04H
     MOV    !_c, a
     MOVW   ax, #07H     ;7
     MOVW   de, #_i
     INCW   DE
     MOV    [DE],A
     DECW   DE
     XCH    A,X
     MOV    [DE],A
     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 RA78K0S assembler. (In this example, the -nca option that differentiates lowercase and uppercase letters must be specified during assembly. -nca is the default.)

```
      NAME ASMSUB

      PUBLIC     _c
      PUBLIC     _i

ABC   DSEG
_c     DB   0
_i     DW   0

      END
```

12.6 Warnings

(1) '_' (underscore)

This C compiler adds an underscore (_ (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.

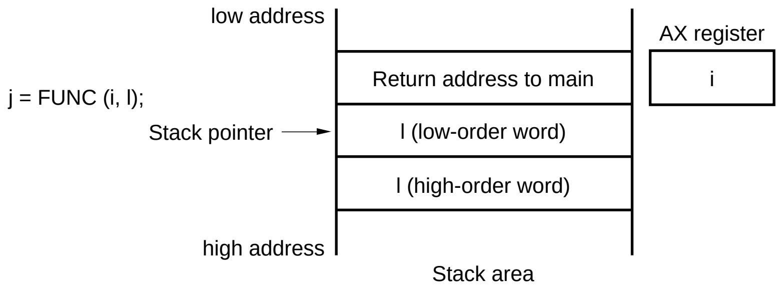
```
extern int FUNC(int, long);     /* Function prototype */

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 RA78K0S.

(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 the Arguments



(3) Run-time library summary

The operation instructions below are called by attaching @@, etc., to the beginning of the function name.

Items for which Δ is indicated in the Support column are supported only in the normal model. If nothing is indicated in the Support column, the item is supported in both the normal and static model.

Table 12-6. Run-Time Library (1/2)

Type	Function Name	Function	Support
Increment	lsinc	Increment signed long.	Δ
	luinc	Increment unsigned long.	Δ
	finc	Increment float.	Δ
Decrement	lsdec	Decrement signed long.	Δ
	ludec	Decrement unsigned long	Δ
	fdec	Decrement float.	Δ
Sign reversal	lsrev	Reverse the sign of signed long.	Δ
	lurev	Reverse the sign of unsigned long.	Δ
	frev	Reverse the sign of float.	Δ
One’s complement	lscom	Determine the one’s complement of signed long.	Δ
	lucom	Determine the one’s complement of unsigned long.	Δ
Logical negation	lsnot	Determine the negative of signed long.	Δ
	lunot	Determine the negative of unsigned long.	Δ
	fnot	Determine the negative of float	Δ
Multiplication	csmul	signed char multiplication	
	cumul	unsigned char multiplication	
	ismul	signed int multiplication	
	iumul	unsigned int multiplication	
	lsmul	signed long multiplication	Δ
	lumul	unsigned long multiplication	Δ
	fmul	float multiplication	Δ
Division	csdiv	signed char division	
	cudiv	unsigned char division	
	isdiv	signed int division	
	iudiv	unsigned int division	
	lsdiv	signed long division	Δ
	ludiv	unsigned long division	Δ
	fdiv	float division	Δ
Remainder	csrem	signed char remainder	
	curem	unsigned char remainder	
	isrem	signed int remainder	
	iurem	unsigned int remainder	
	lsrem	signed long remainder	Δ
	lurem	unsigned long remainder	Δ
Addition	lsadd	signed long addition	Δ
	luadd	unsigned long addition	Δ
	fadd	float addition	Δ

Table 12-6. Run-Time Library (2/2)

Type	Function Name	Function	Support
Subtraction	lssub	signed long subtraction	Δ
	lusub	unsigned long subtraction	Δ
	fsub	float subtraction	Δ
Left shift	lslsh	Left shift of signed long	
	lulsh	Left shift of unsigned long	
Right shift	lsrsh	Right shift of signed long	
	lursh	Right shift of unsigned long	
Comparison	cscmp	signed char comparison	
	iscmp	signed int comparison	
	lscmp	signed long comparison	Δ
	lucmp	unsigned long comparison	Δ
	fcmp	float comparison	Δ
Bit AND	lsband	Bit AND of signed long	Δ
	luband	Bit AND of unsigned long	Δ
Bit OR	lsbor	Bit OR of signed long	Δ
	lubor	Bit OR of unsigned long	Δ
Bit XOR	lsbxor	Bit XOR of signed long	Δ
	lubxor	Bit XOR of unsigned long	Δ
Logical AND	fand	Logical AND of float	Δ
Logical OR	for	Logical OR of float	Δ
Conversion from floating point	ftols	Converts from float to signed long	Δ
	ftolu	Converts from float to unsigned long	Δ
Conversion to floating point	lstof	Converts from signed long to float	Δ
	lutof	Converts from unsigned long to float	Δ
Conversion from bit	btol	Converts bit to long	Δ
Start-up routine	cstart	Required preparation for system execution	
Function pre-and postprocessing	cprep	Function preprocessing	Δ
	cdisp	Function postprocessing	Δ
	hdwinit	Initialization of peripheral equipment (sfr) after CPU reset	
Error check	chkstk	Check for stack overflow	
Error processing	errstk	Error processing routine for stack overflow	
Bank Function	fcall	Call a bank Function	
BCD type conversion	bcdtob	Convert from 1-byte bcd to 1-byte binary	
	btobcd	Convert from 1-byte binary to 2-byte bcd	
	bcdtow	Convert from 2-byte bcd to 2-byte binary	
	wtobcd	Convert from 2-byte binary to 2-byte bcd	

The run-time library does not check for errors even when the -L option is specified during compilation. The library does not support computations not shown above. The compiler performs inline expansion. The long addition and subtraction; and, or, and xor; and shifting are sometime expanded inline.

[MEMO]

CHAPTER 13 EFFICIENT COMPILER USE

This chapter explains how to efficiently use this C compiler.

13.1 Command Input When Compiling

In this C compiler, the type of the target device is specified when compiling the source file. This specification of the device type can be described in the C source.

- Macro name indicating the device series name
  '_ _K0S_ _'
- Macro name indicating the device name
  Two underscores '_ _' are added to the beginning of the device type name and one '_' to the end.
  Specify uppercase letters.

  (Example) _ _054_ _ _054Y_

The device type is specified during compilation as follows.
The following specification is added to the command line when compiling.

```
-c device-type-name
```

***** Refer to the product data sheet for the device file being used for information about the device type names.

(Example) cc78K0S -c9024 prime.c

By specifying the device type in the C source, this specification becomes unnecessary during compilation.
This statement is specified at the beginning of the C source program.

```
#pragma PC(type)
```

(Example) #pragma PC(9024)

The following can be described before "#pragma PC(type)".

- Comments
- Preprocessing directives that do not generate variable or function definitions or references

13.2 Efficient Coding

When developing a 78K/0S Series application product, efficient objects can be created by using the saddr area or callt area of the device in the C compiler.

```
• Using external variables
   |
   └── if ( saddr area can be used ) ──── sreg / _ _sreg variables are used /
                                          Compiler option (-qd) is used.

• Using one-bit data
   |
   └── if ( saddr area can be used ) ──── bit / boolean / _ _boolean type variables are used.

• Defining functions
   |
   ├── if ( functions with many call locations)
   |        |
   |        └── if ( callt area can be used)
   |                 |
   |                 └── Make it a _ _callt or callt function ( effective in reducing the code size)
   |
   ├── if ( frequently called functions)
   |        |
   |        └── if (not used recursively)
   |                 |
   |                 └── Make it a _ _leaf or norec function ( effective in improving the
   |                     execution speed and in reducing the code size)
   |
   ├── if ( automatic variables are not used )
   |                 |
   |                 └── Make it a noauto Function (effective in improving the execution speed
   |                     and in reducing the code size)
   |
   |   if ( automatic variables are used && saddr area can be used )
   └──               |
                     └── Declare with register (effective in improving the execution speed and
                         in reducing the code size)
```

(1) Defining external variables

If the saddr area can be used when defining external variables, the external variables to be defined become sreg or _ _sreg variables. sreg or _ _sreg variables reduce the instruction code, can reduce the object code, and improve the execution speed compared to instructions for memory. (The same effect occurs when using the -qd option instead of sreg variables.)

```
sreg/_ _sreg variable definition :    extern sreg int variable-name;
                                      extern _ _sreg int variable-name;
```

***** See Section 11.4 “(3) Using saddr area.”

(2) 1-bit data

Objects that use only 1-bit data become bit type variables (or boolean or __boolean type variables). Bit manipulation instructions are generated in operations on bit or boolean, or __boolean type variables.

```
bit/boolean type variable declarations :  bit variable-name;
                                          boolean variable-name;
                                          _ _boolean variable-name;
```

* See Section 11.4 “(7) bit type variable.”

(3) Function definition

- Functions with many call locations
  Functions that have many call locations and can use the callt area are callt functions. Callt functions are called using the callt table area of the device, therefore their object code is shorter than that of regularly called functions. (* Definition of callt function: callt int tsub*() “11.4 (1) callt function”)

- Frequently called functions
  A function that is called frequently must reduce the object code or make a structure which can be called at thigh speed. Consequently, frequently called function which is not used recursively becomes a norec funtion. A norec funtion becomes are without pre- or post-processing (stack frame). Due to this definition the object code can be reduced and the execution speed can be improved in comparision with a normal function. (*norec function definition: norec int rout() in the section 11.4 norec funtion)

- Functions that do not use automatic variables
  Functions that do not use automatic variables become noauto functions. A noauto function does not have a stack frame. Arguments can be passed by register when possible. The object code can be reduced and the execution speed improved.
  Even in a function where noauto is not declared, by using register declarations of the arguments without using automatic variables, the same function is obtained as a noauto function.

  ***** noauto function definition : noauto int sub1(int i) “11.4 (5) noauto function”
  register variable definition : int sub1(register int i) “11.4 (2) Register variables”

- Functions that use automatic variables

  If the saddr area can be used in a function that uses automatic variables, register is declared. A register declaration allocates the declared object to the microprocessor's registers. A program that used registers operates faster than a program that uses memory and can also have compact object code.

  ***** register variable definition : register int i; "11.4 (2) Register variables"

APPENDIX A saddr AREA LABEL SUMMARY

In the CC78K0S, the saddr area is referenced by the following labels. Therefore, the same names as the labels cannot be used in the C source program or assembler source program. (Refer to Table 11-2, "Memory Space Use".)

[Normal Model]

(1) Register variables

Label	Address
_@KREG00	0FED8H
_@KREG01	0FED9H
_@KREG02	0FEDAH
_@KREG03	0FEDBH
_@KREG04	0FEDCH
_@KREG05	0FEDDH
_@KREG06	0FEDEH
_@KREG07	0FEDFH
_@KREG08	0FEE0H
_@KREG09	0FEE1H
_@KREG10	0FEE2H
_@KREG11	0FEE3H
_@KREG12	0FEE4H*****
_@KREG13	0FEE5H*****
_@KREG14	0FEE6H*****
_@KREG15	0FEE7H*****

***** When the function argument undergoes the register declaration and the -QR option is specified, an argument is allocated in the saddr area.

(2) norec function arguments

Label	Address
_@NRARG0	0FEE8H
_@NRARG1	0FEEAH
_@NRARG2	0FEECH
_@NRARG3	0FEEEH

(3) norec function automatic variables

Label	Address
_@NRAT00	0FEF0H
_@NRAT01	0FEF1H
_@NRAT02	0FEF2H
_@NRAT03	0FEF3H
_@NRAT04	0FEF4H
_@NRAT05	0FEF5H
_@NRAT06	0FEF6H
_@NRAT07	0FEF7H

(4) Run-time library arguments

Label	Address
_@RTARG0	0FEF8H
_@RTARG1	0FEF9H
_@RTARG2	0FEFAH
_@RTARG3	0FEFBH
_@RTARG4	0FEFCH
_@RTARG5	0FEFDH
_@RTARG6	0FEFEH
_@RTARG7	0FEFFH

[Static Model]

(1) Common area

Label	Address
_@KREG00	0FEF0H
_@KREG01	0FEF1H
_@KREG02	0FEF2H
_@KREG03	0FEF3H
_@KREG04	0FEF4H
_@KREG05	0FEF5H
_@KREG06	0FEF6H
_@KREG07	0FEF7H
_@KREG08	0FEF8H
_@KREG09	0FEF9H
_@KREG10	0FEFAH
_@KREG11	0FEFBH
_@KREG12	0FEFCH
_@KREG13	0FEFDH
_@KREG14	0FEFEH
_@KREG15	0FEFFH

[MEMO]

APPENDIX B SEGMENT NAMES

All of the segments output by the compiler are described.

(1) C source module example (SAMPLE.C)

```
#pragma INTERRUPT INTP0 inter            /* Interrupt function */
void main(void)                          /* Function prototype */
const int i_cnst = 1;                    /* const variable */
callt int f_clt(void);                   /* callt function prototype */
boolean b_bit;                           /* boolean type variables */
long l_init = 2;                         /* Initialized external variable */
int i_data;                              /* Uninitialized external variable */
sreg int sr_inis = 3;                    /* Initialized sreg variable */
sreg int sr_dats;                        /* Uninitialized sreg variable */

void main()                              /* Function definition */
{
          int i;
          i = 100;
}

void inter()                             /* Interrupt function definition */
{
          b_bit = 1;
}

callt int f_clt()                        /* callt function definition */
{
}
```

(2) Assembler source module Example

(1) When ROM is specified

```
;78K/0S Series C Compiler Vx.xx Assembler Source
;                                                 Date:xx xxx xxxx Time:xx:xx:xx
;Command   : -c9024 sample.c -a -ng
;In-file   : SAMPLE.C
;Asm-file  : SAMPLE.ASM
;Para-file:

$PROCESSOR (9024)
$NODEBUG
$NODEBUGA
$SYMLEN
$NOCAP
$KANJICODE SJIS
$TOL_INF     03FH, 0100H, 02H, 00H

         EXTRN  _@cprep
         PUBLIC _inter
         PUBLIC _main
         PUBLIC _i_cnst
         PUBLIC ?f_clt
         PUBLIC _b_bit
         PUBLIC _l_init
         PUBLIC _i_data
         PUBLIC _sr_inis
         PUBLIC _sr_dats
         PUBLIC _f_clt

@@BITS       BSEG                          /* bit type variable segment */
_b_bit   DBIT

@@CNST       CSEG                          /* const variable segment */
_i_cnst: DW  01H   ;1

@@R_INIT     CSEG                          /* Initialized data segment (initial value) */
         DW  00002H,00000H ;2

@@INIT       DSEG                          /* Data area segment (initial value) */
_l_init: DS  (4)

@@DATA       DSEG                          /* Data area segment (no initial value) */
_i_data: DS  (2)
```

```
@@R_INIS      CSEG UNITP                          /* Initialized data segment (initialized sreg variable) */
         DW   03H ;3

@@INIS        DSEG SADDRP                         /* Data area segment (initialized sreg variable) */
_sr_inis:DS   (2)

@@DATS        DSEG  SADDRP                        /* Data area segment (uninitialized sreg variables */
_sr_dats:     DS    (2)

@@CALT        CSEG  CALLT0                        /* callt function segment */
?f_clt:  DW   _f_clt

@@CODE        CSEG                                /* Code segment */
_main:
       push  hl
       movw  ax, #02H
       callt  [_@cprep]
       movw   ax, #064H     ;100
       mov    [hl+1],a    ;i
       xch    a, x
       mov    [hl],a      ;i
       pop    ax
       pop    hl
       ret
_inter:
       set1  _b_bit
       reti
_f_clt:
       ret

@@VECT06      CSEG  AT    0006H                   /* interrupt vectored table */
       DW    _inter
       END

; Target chip :  μPD789024
; Device file  :  Vx.xx
```

(2) When not programmed in the ROM

```
;78K/0S Series C Compiler Vx.xx Assembler Source
;                                                Date:xx xxx xxxx Time:xx:xx:xx
;Command  : -c9024 sample.c -a -ng -nr
;In-file  : SAMPLE.C
;Asm-file : SAMPLE.ASM
;Para-file:

$PROCESSOR (9024)
$NODEBUG
$NODEBUGA
$SYMLEN
$NOCAP
$KANJICODE SJIS
$TOL_INF      03FH,0100H,02H,00H

         EXTRN  _@cprep
         PUBLIC _inter
         PUBLIC _main
         PUBLIC _i_cnst
         PUBLIC ?f_clt
         PUBLIC _b_bit
         PUBLIC _l_init
         PUBLIC _i_data
         PUBLIC _sr_inis
         PUBLIC _sr_dats
         PUBLIC _f_clt

@@BITS       BSEG                             /* bit type variable segment */
_b_bit   DBIT

@@CNST       CSEG                             /* const variable segment */
_i_cnst: DW  01H   ;1


@@INIT       DSEG                             /* Data area segment (initial value) */
_l_init: DW  00002H,00000H  ;2

@@DATA       DSEG                             /* Data area segment (no initial value) */
_i_data: DB  (2)

@@INIS       DSEG  SADDRP                     /* Data area segment (initialized sreg variable) */
_sr_inis:DW  03H    ;3

@@DATS       DSEG  SADDRP                     /* Data area segment (uninitialized sreg variable) */
_sr_dats:    DB    (2)
```

```
@@CALT       CSEG  CALLT0                          /* callt function segment */
?f_clt:  DW  _f_clt

@@CODE       CSEG                                  /* Code segment */
_main:

         push  hl
         movw  ax, #02H
         callt [_@cprep]
         movw  ax, #064H     ;100
         mov   [hl+1], a  ;i
         xch   a, x
         mov   [hl], a    ;i
         pop   ax
         pop   hl
         ret
_inter:
         set1  _b_bit
         reti
_f_clt:
         ret

@@VECT06    CSEG  AT   0006H                       /* interrupt vectored table*/
         DW _inter
         END

;Target chip :  μPD789024
;Device file  :  Vx.xx
```

[MEMO]

APPENDIX C FUNCTION INTERFACE LIST

C.1 Storage Locations of Return Values

Type	Normal model	Static model
1-byte, 2-byte integer	BC	AX
4-byte integer	BC (low-order), DE (high-order)	AX (low-order), BC (high-order)Note 1
Pointer	BC	AX
Structure, union	BC (Start address of structure, 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

Note 1. Not supported in V1.00.

C.2 Passed on Argument (Function Call Side)

C.2.1 Normal Model

Type	Location to be passed (1st argument)	Location to be passed (2nd argument or later)
1-byte, 2-byte data	AX	Passed on stack
4-byte data	AX, BC	Passed on stack
Floating-point number	AX, BC	Passed on stack
Other	Passed on stack	Passed on stack

* 1- to 4-bye data includes structure and union.

C.2.2 Static Model

Type	Location to be passed (1st argument)	Location to be passed (2nd argument)	Location to be passed (3rd argument)
1-byte data	A	B	H
2-byte data	AX	BC	HL
4-byte data	Allocate this data to AX and BC, and the remains to H or HL.		

* 1- to 4-bye data includes structure and union.

C.3 Argument/Automatic Variables Storage List (Called Function)

C.3.1 Normal Model

Option	Argument/auto variable	Storage location	Priority level
-QV (register allocation option)	Declared argument or automatic variable	HL register (only if base pointer is not required)	char type: L, H, in this order int, short, enum type: HL
-QR (saddr allocation option)	Declared argument or automatic variable (including register variable)	Argument: _@KREG12 to 15 [0FEE4H to 0FEE7H] auto variable: _@KREG00 to 11 [0FED8H to 0FEE3H] _@KREG12 to 15 is not allocated to argument	Only number of bytes of variable or argument is allocated, in order of appearance
-QRV	Declared argument or automatic variable (including register variable)	HL register, Argument: _@KREG12 to 15 [0FEE4H to 0FEE7H] auto variable: _@KREG00 to 11 [0FED8H to 0FEE3H], _@KREG12 to 15 if not allocated to argument	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	In order of appearance

C.3.2 Static Model

Option	Argument/auto variable	Storage location	Priority level
-QV (register allocation option)	Declared argument or automatic variable	DE register	char type: L, H, in this order int, short, enum type: HL
Default	Declared argument or automatic variable	Function-specific area	Arguments are allocated in reverse order of appearance. Automatic variables are allocated in order of appearance.
Default	Arguments with register declaration Register variables	DE regisger	Only number of bytes of variable or argument is allocated in order of appearance. Number of bytes or more are allocated in the function-specific area.

INDEX

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America NEC Electronics Inc. Corporate Communications Dept. Fax: 1-800-729-9288	**Hong Kong, Philippines, Oceania** NEC Electronics Hong Kong Ltd. Fax: +852-2886-9022/9044	**Asian Nations except Philippines** NEC Electronics Singapore Pte. Ltd. Fax: +65-250-3583
Europe NEC Electronics (Europe) GmbH Technical Documentation Dept. Fax: +49-211-6503-274	**Korea** NEC Electronics Hong Kong Ltd. Seoul Branch Fax: 02-528-4411	**Japan** NEC Corporation Semiconductor Solution Engineering Division Technical Information Support Dept. Fax: 044-548-7900
South America NEC do Brasil S.A. Fax: +55-11-889-1689	**Taiwan** NEC Electronics Taiwan Ltd. Fax: 02-719-5951	

I would like to report the following error/make the following suggestion:

Document title: ______________________________

Document number: ______________________ Page number: __________

____

____

____

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	❏	❏	❏	❏
Technical Accuracy	❏	❏	❏	❏
Organization	❏	❏	❏	❏

CS 96.4