To our customers,

Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

Notice

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

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

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

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

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

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

User's Manual

32

M3T-CC32R V.4.30

User's Manual <C Compiler>

Cross Tool Kit for M32R Family

Renesas Electronics
www.renesas.com

Rev.1.00 2004.09

Keep safety first in your circuit designs!

- Renesas Technology Corporation and Renesas Solutions Corporation put the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

- These materials are intended as a reference to assist our customers in the selection of the Renesas Technology product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation, Renesas Solutions Corporation or a third party.
- Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
- All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation and Renesas Solutions Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor for the latest product information before purchasing a product listed herein. The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors. Please also pay attention to information published by Renesas Technology Corporation and Renesas Solutions Corporation by various means, including the Renesas home page (http://www.renesas.com).
- When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability or other loss resulting from the information contained herein.
- Renesas Technology semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
- The prior written approval of Renesas Technology Corporation and Renesas Solutions Corporation is necessary to reprint or reproduce in whole or in part these materials.
- If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
- Please contact Renesas Technology Corporation or Renesas Solutions Corporation for further details on these materials or the products contained therein.

For inquiries about the contents of this document or product, fill in the text file the installer generates in the following directory and email to your local distributor.

¥SUPPORT¥Product-name¥SUPPORT.TXT

Renesas Tools Homepage http://www.renesas.com/

Contents

Preface x

Chapter 1 Overview of CC32R 1

Chapter 2 Overview of cc32R 5

Chapter 3 Invoking the Compiler 9

Chapter 6 C Calling Conventions 78

Chapter 7 Embedded Applications Programming 94

Chapter 8 Standard Header Files 116

Chapter 9 C Standard Library 138

Chapter 10 The cc32R's Behavior 280

Preface

M3T-CC32R(abbreviated as CC32R) is a cross tool kit which supports software development for the Renesas M32R family of 32-bit RISC architecture microcomputers. It provides many functions suitable for development of embedded systems for the M32R family. The CC32R manual set provides information for programming by use of CC32R, targeting an M32R system.

Audience

The CC32R manual set assumes that the readers are developers programming for the M32R system using the C or assembly language. Accordingly, it also assumes that the readers are familiar with programming languages (C or assembly) and their development environment (a host machine and its operating system etc.), and have basic knowledge of the target M32R systems.

References

A manual related to development for the M32R family is :

- M32R Family User's Manual
- M32R Family Software Manual

Refer to the WWW site of the "Renesas Microcomputers" .

The URL is : http://www.renesas.com/

For details about the ANSI-C language, refer to :

- ANSI/ISO9899-1990 AmericanNationalStandardforProgrammingLanguages-C (AmericanNationalStandardsInstitute,Inc.)

Conventions

The CC32R manual set uses the following conventions :

- Symbols

Symbol	Meaning
Italics	Represents a generic description that should be replaced with a specific.
a | *b*	Represents alternative items. *a* | *b* represents either *a* or *b*.
[ ]	Encloses optional elements that can be included or omitted.
...	Indicates to repeat the preceding item zero or more times.
:	Represents omission of a or more lines.
`<RET>`	Represents to enter the return key.

- Terms(1/2)

Term	Meaning
ANSI-C	American National Standard for Programming Languages-C (ANSI/ISO 9899-1990)
Assembler (as32R)	The assembler in CC32R.
Assembly program	A program written in the assembly language.
CC32R	The cross tool kit for an M32R system.
C compiler (cc32R)	The C compiler in CC32R.
C program	A program written in the C language.
CRx	Any control register of M32R.
C standard library	The CC32R-supplied ANSI-C conforming library.
Default	A value (or values) or the process provided automatically if there is none specified by the user.
EWS	An engineering work station.
Librarian (lib32R)	The librarian in CC32R.
Library (file)	A C library file for an M32R system. It is an output file from lib32R.
Linker (lnk32R)	The linker in CC32R.

- Terms(2/2)

Term	**Meaning**
Link map	A list have information on sections and global symbols in an object module or a load module. It is generated by map32R.
Load module (file)	A linked object module, which is an executable file for an M32R system. It is an output file from lnk32R or lmc32R.
Load module converter (lmc32R)	The load module converter in CC32R.
Local variable	This variable is only effective in a function.
M32R M32Rx	A Renesas 32-bit RISC architecture microcomputer.
M32R system	A system using the M32R.
Map generator (map32R)	The map generator in CC32R.
Object module (file)	An object file which is translated from the C or assembly code into the object code of machine instructions for M32R. It is an output file from the C compiler or the assembler.
OS	An operating system.
Release notes	The document related to the release of the CC32R in a CC32R package (Please read it at first.).
Return value	A function value returned as an operation result from a called function to a calling function.
Rx	Any general register of M32R.
Source file	A text file written source code in the C language or the assembly language.
Space (character)	A blank which is entered by the space key or the tab key.
User library	A library file made by a user using the librarian.
Windows	Any of Microsoft Windows3.1 or Microsoft Windows95.

Chapter 1

Overview of CC32R

Features

CC32R is a cross tool kit designed to develop an application program for the M32R family. Its versatile features are especially useful for development of a control system to be embedded. Major features of the CC32R include the following :

- Generates a load module file which is executable on the M32R from a source file written in C or assembly language.
- Converts a load module into S-format one which can be written into ROM.
- Offers optimizing functions which generate efficient object codes to speed execution time.

Components of CC32R

CC32R consists of the following cross tools :

- C complier (cc32R) with C standard libraries
- Assembler (as32R)
- Linker (lnk32R)
- Librarian (lib32R)
- Map generator (map32R)
- Load module converter (lmc32R)

Overview of the Components of CC32R

The following tools are contained in CC32R :

- C compiler (cc32R)

  The C compiler cc32R conforms with ANSI/ISO 9899-1990. It generates an assembly source file by compiling a C source file. As a driver, it also invokes the assembler and/or the linker. You can perform a chain process as creating a load module from a source file.

- Assembler (as32R)

  The as32R generates an object module by assembling an assembly source file. In an assembly source file, you can write the pseudo-instructions and the macro-instructions. Also, the assembler outputs an assemble list with the -l option.

- Linker (lnk32R)

  The lnk32R generates a load module file by linking an object module, the relocatable load module files and library files. It is selectable that either a generated load module is relocatable format or absolute format. A relocatable load module can be reloaded onto the linker.

- Librarian (lib32R)

  The lib32R generates an M32R compatible library file from object modules or relocatable load modules.

- Map generator (map32R)

  The map32R outputs a link map which consists of a map list and global symbol list from an object module, an relocatable load module and an absolute load module.

- Load module converter (lmc32R)

  The lmc32R converts a load module generated at the linker into the S-format load module (ROMable). This tool is necessary to write a program into ROM.

Programming Flow

The programming flow using CC32R is shown in Figure 1.1.

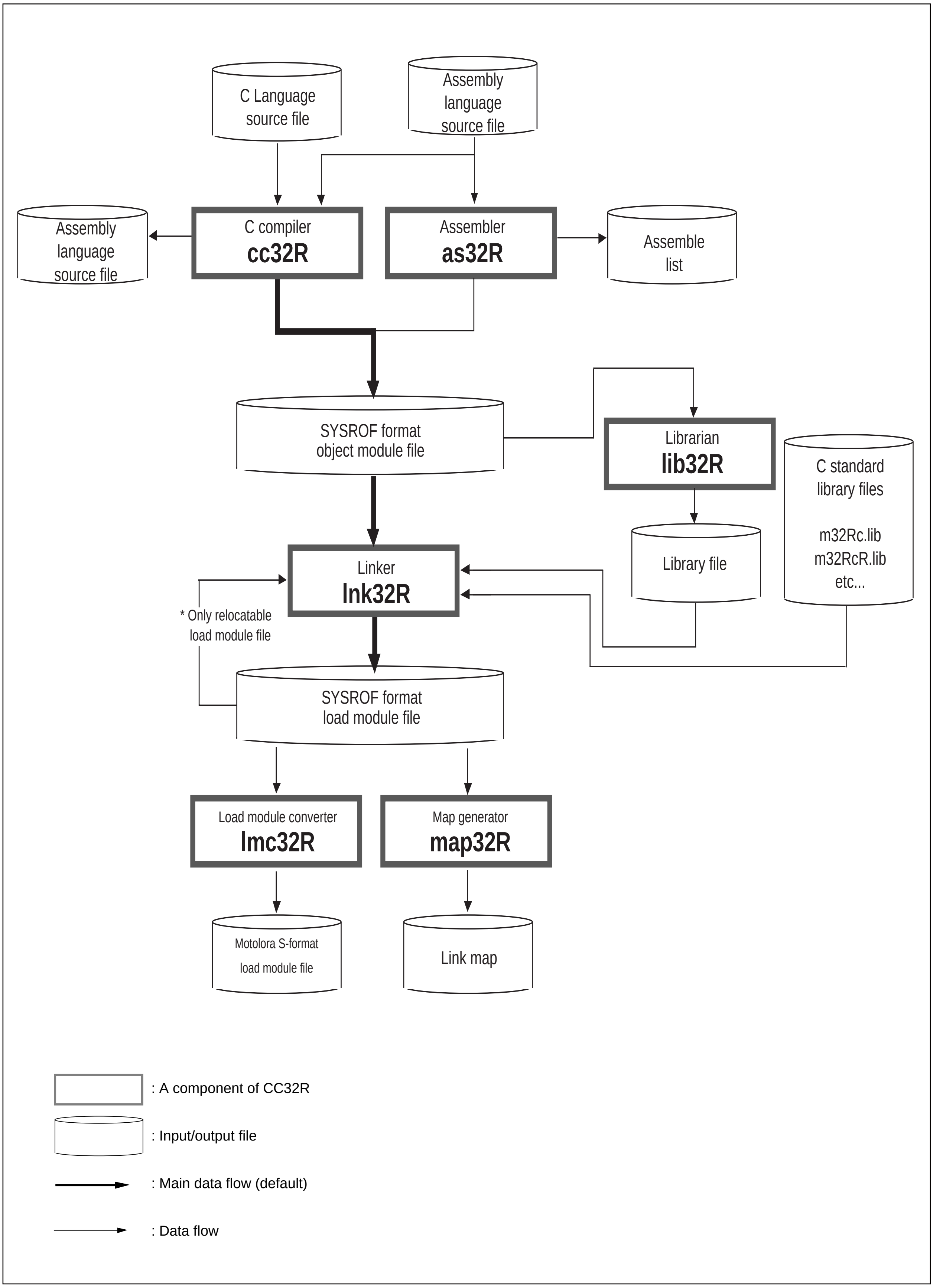

C Language source file
Assembly language source file
Assembly language source file
C compiler cc32R
Assembler as32R
Assemble list
SYSROF format object module file
Librarian lib32R
C standard library files
m32Rc.lib
m32RcR.lib
etc...
Library file
Linker lnk32R
* Only relocatable load module file
SYSROF format load module file
Load module converter lmc32R
Map generator map32R
Motolora S-format load module file
Link map
: A component of CC32R
: Input/output file
: Main data flow (default)
: Data flow

Figure 1.1 Programming Flow by CC32R

Input and Output File Names for CC32R

CC32R identifies the type of an input file by its file name extension. And CC32R will determine the file name or the extension for an output file. Table 1.1 lists the input/output file names that are handled by CC32R. *file* represents any file name.

Table 1.1 File names for CC32R

File	Type
file`.c`	C source file
file`.mi`	Preprocessor output file (i.e., a C source file after expansion by the preprocessor)
file`.ms`	Assembly source file
file`.mo`	Object module file
`a.mout`	Load module file
file`.mot`	Load module file (Motorola S-format)

Chapter 2

Overview of cc32R

2.1 About the C Compiler cc32R

2.1.1 cc32R Functions

The cc32R is the C complier contained in the CC32R cross tool kit, and has the following functions :

- Generates an assembly source file by compiling each C source file (specified by the -S option).
- Generates an object module file from each C or assembly source file (as default or specified by the -c option).
- Generates a load module file from C and/or assembly source files. This function is the default. By default, cc32R performs a chain process as compiling, assembling, and linking. Contents of each source file (e.g., a use language) are discriminated by the filename extension.

2.1.2 cc32R Features

- Conformance with ANSI

  The C complier and the C standard libraries conform with the ANSI Standard, ANSI/ISO 9899-1990.

- The set of 64-bit integer arithmetic functions is supported

  The set of functions to perform C language integer arithmetic in the 64-bit dynamic range has been added to the standard library. As for the integer type in C language, these functions can perform the four fundamental operations in arithmetic, as well as bitwise, shift and compare operations in the 64-bit range.

- Floating-point operation is supported

  Internal data representation of floating-point data conforms with the IEEE (The Institute of Electrical and Electronics Engineers) 754 Standard.

- Creating ROMable programs is supported

  ROMable object modules can be created (This depends on the linker's functions.). When linking and placing sections of object modules, space can be allocated for the section which contains initialized data within the RAM area, and the initialized data can be placed in the ROM area.

- Supports optimization function

  Optimization function enables efficient generation of object code. The following optimization levels are supported :

  - Optimization at assembly language :
    Eliminates unnecessary codes at assembly language level, converts to most suitable codes and schedules instructions.

  - Local optimization :
    Analyzes C language structure locally, transfers or copies constants and deletes unnecessary codes and common sub-expressions.

  - Global optimization :
    Analyzes C language structure globally, analyzes live variables, replaces codes and optimizes control flow.

  The following lists the optimization items performed:

  - Optimize control flow
  - Delete expressions in common parts
  - Propagate constants and copy
  - Analyze valid variables
  - Delete dead code
  - Optimize register assignment

  The combination of different optimization levels further improves efficiency of code generation although each optimization can be used independently.

  Most optimizations combined at different levels increase program execution speed and reduce code size. However, some optimizations offer only one advantage or offer better functional performance at the sacrifice of some other advantage. When using these optimization functions, the user should specify which optimization advantage should come first.

- Selectable output file

  The result of compiling can be output in the form of an assembly language source file, an object module file or a load module file (linked object module). The assembly language source file is useful when checking the C language source program at the assembly language level.

- Output of C language source line debug information

  Information on debugging of C language source line can be added to the load module generated by the linker (with the -g option selected). By using the load module, the debugger can be used for debugging at the C language source line level.

- C++ like comments can be written.

  A comment can be written starting from "//" to the end of line.

```
Example:
void foo(void)
{
   char   x[3];
   x[0] = 0;  // Comments can be written in this manner
}
```

- Development Support Utilities

  ❍ **C source merge utility strip32R**

  We provide the C source merge utility **cmerge.**
  The utility cmerge merges the assembler source file (with -CS option added) that is output by the compiler with the original C source file to generate a C/ASM-mixed list.
  The utility strip32R is not a standard product of the compiler. For details on how to handle it, refer to the file written about the Handling of Development Support Utilities.For more information of strip32R, refer to the certain documents in directory "UnSpt32R" (PC version) or "unsupport" (EWS version). The "license.txt" explains about license, and the "strip32R.txt" explains about how to use it.

  ❍ **Absolute listing utility abslist**

  We provide the absolute listing utility **abslist**.
  The utility abslist generates an absolute list file in which the LOCATION values in the assembly list files output by the assembler (as32R) have been converted into the actual addresses after linking.
  The utility abslist is not a standard product of the compiler. For details on how to handle it, refer to the file written about the Handling of Development Support Utilities. For details on how to use it, refer to the file in which a description is made of Absolute Listing Utility (abslist.sj).

  ❍ **Stack size calculation utility stk32R**

  We provide the Stack size calculation utility **stk32R.**
  The utility stk32R processes the stack amount usage files output by the compiler (-stack option added) to find the stack size required for program operation.Also available are the stack amount usage files (m32Rc.stk, m32RcR.stk, etc.) for the functions registered in C standard libraries(m32Rc.lib, m32RcR.lib, etc.). To enter the stack amount usage file for the program that calls library functions in stk32R, specify in -I option the stack amount usage files present in the library file used. Note that correct values cannot be obtained for the

functions pow and setlocale because these are recursive-call functions. The values in the files are for reference purposes only.
The utility stk32R is not a standard product of the compiler. For details on how to handle it, refer to the file written about the Handling of Development Support Utilities. For details on how to use it, refer to the file in which a description is made of Stack Size Calculation Utility (stk.txt).

2.2 Compatibility with an old version

- About inputting old CC32R's object (V.2.10 Release 1 or older) to new linker

To correspond to the new function, a part of object format has been changed.Accordingly, if you have the object that was made with old CC32R (V.2.10 Release 1 or older), when you input them to new linker (CC32R V.3.00 Release 1 or newer), this linker displays a warning message like the following.

```
lnk32R: " filename": warning: old interface module: "revision:01"
```

In this case, please remake these objects by using the new CC32R.

- Problems encountered when linking objects of V.1.00 Release 3 or earlier

An error "relocation out of range " may be encountered when linking some objects generated by CC32R V.1.00 Release 3 or earlier by the linker in V.1.00 Release 4 or later (including this version). In such a case, regenerate the objects using the assembler in V.1.00 Release 4 or later.

Chapter 3

Invoking the Compiler

3.1 How to Invoke the Compiler

3.1.1 Invoking Procedure

To invoke the C compiler, first set the environment variables (see 3.1.2), enter the “cc32R” command according to the command line rules and execute it (see 3.1.3).

3.1.2 Setting Environment Variables

Set the valid directories for the environment variables M32RBIN, M32RINC, M32RLIB and M32RTMP. (This step may be skipped since these variables are normally set during installation.) For the setting procedure, refer to “ CC32R Installation Guide”. If you do not set the directories, the default directories are selected automatically.

Table 3.1 Environment Variables

Environment Variable	Default	
M32RBIN	/usr/local/M32R/bin	
M32RINC	/usr/local/M32R/include	
M32RLIB	/usr/local/M32R/lib	
M32RTMP	/tmp	
M32RKIN	EWS version “euc”	MS-Windows(PC) version “sjis”
NR32RKOUT	EWS version “euc”	MS-Windows(PC) version “sjis”

3.1.3 Command Line Syntax and Rules

The command line syntax and rules for the command, “cc32R”, which invokes the C compiler are as follows (For details on the command options and input/ output files, see 3.1.4 to 3.2.) :

```
    cc32R   [-access=access_control_file] [-c] [-C]
            [-constr] [-D name[=def]] [-e entrypoint] [-E]
            [-float_only] [-fminst] [-g] [-I path] [-L dir]
            [-l lib] [-M] [-MAP map_filename]
            [-MEM addr1,addr2] [-noinline]
            [-o output_filename] [-Opriority]
            [-O[level]] [-m32re5] [-P] [-r] [-R old=new] [-S]
            [-SEC name[=addr][,name[=addr]...]]
            [-switch_by_offset] [-U name] [-v] [-V] [-w]
            [-warn_suppressed_nested_comment] [-rel16]
            [-XX[=symbol_num]]
            [-small [-memlarge]] [-medium] [-large]
            [-CS] [-stack] [-zdiv][-@]
            [input_filenames] <RET>
```

where :

- Without [ ] : Indispensable
- In [ ] : Optional
- Prefixed by - : A command option (see 3.2)
- <RET> : Enter the return key

Figure 3.1 cc32R Command Line Syntax

- Write into the command line by following the format shown in Figure 3.1. Each of the items (i.e., the command name, an option, an input file name) must be separated from adjacent items by at least one space character. By entering the return key at the end of a command line, the C complier starts execution.

- Between an option and its parameter(s), one or more spaces may be inserted.

- *input_filenames* represents the specification of one or more input file names. Between input file names, one or more spaces are needed for separation. The number of files is not limited.

- The type of each input file is determined based on its extension as listed in Table 3.2 :

Table 3.2 Input File Name and Type

Extension	Type
.c	C source file
.ms	Assembly source file
.mo	Object module file
Others	Object module file

3.1.4 Command file

When invoking CC32R,one or more command options listed in a command file (a text file) can be specified by one parameter.

```
   cc32R   @file_name [-@] <RET>
```

where :
- *file_name* : File name of Command file
- In `[ ]` : Optional
- `<RET>` : Enter the return key

Figure 3.2 cc32R Command file Syntax

- As a parameter, specify only a command file name prefixed by `'@'`. If anything other than one "*@command_filename*"(except for the -@ option) is specified, it will not assumed as a command file even if its beginning character is '@'.

  Example 1 : `@sample.cmd` processed as a command file :

  >cc32R @sample.cmd
  >cc32R -@ @sample.cmd
  >cc32R @sample.cmd -@

  Example 2 : `@sample.cmd` is not assumed as a command file :

  >cc32R -v @sample.cmd —— there is a parameter which is not command file specification (except for -@) .
  >cc32R @sample.cmd -v —— ditto.
  >cc32R @sample.cmd @sample.cmd —— there are two or more command file specifications.

- Rules for the command file are :

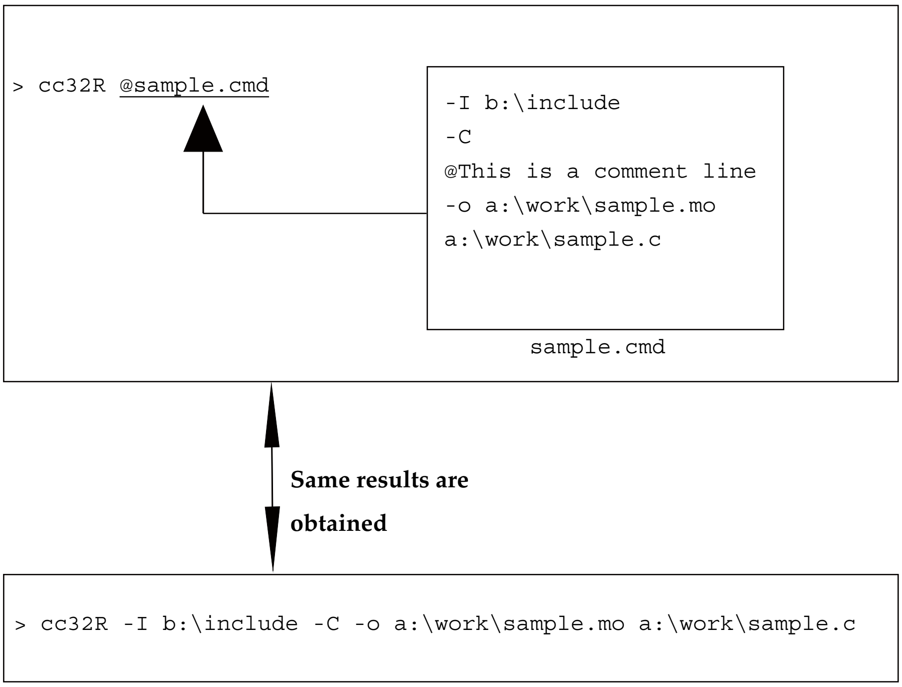
  O Each parameter (options, input/output filenames, etc.) takes the same format as parameters specified on the command line.

  O Delimit parameters with one or more spaces or a new-line character (return key).

  O Lines starting with @ are seen as comments and are skipped.

  O You cannot call a command file from within a command file.

Figure 3.3 shows results when commands are executed from a command file and from the command line in ordinary format.



```
> cc32R @sample.cmd
```

```
-I b:\include
-C
@This is a comment line
-o a:\work\sample.mo
a:\work\sample.c
```

```
> cc32R -I b:\include -C -o a:\work\sample.mo a:\work\sample.c
```

Figure 3.3 When executing commands from a command file and from the command line

||||| NOTE! |||||

lnk32R (linker) and lib32R (librarian) process files as command files even when the filename does not start with ‘@’. Note, however, that lines starting with ‘@’ in such command files are not processed as comments.

||||| NOTE! |||||

Do NOT use filenames starting with ‘@’ other than for command files, as command lines will not be processed correctly (operation cannot be assured).

3.1.5 Input File Conditions

Conditions of the input files which can be processed on the compiler are listed in Table 3.3. If a file does not meet these conditions, you should not input the file.

Table 3.3 Input File Conditions

Where	Conditions
Valid input files	C source file(s) or M32R assembly source file(s) Object module file(s) Load module file(s) Library(s)
Valid name length	Identifiers(function name, variable name and etc...) : Up to 240 characters Note: The different names as if the initial parts which consists of characters from the 1st to the 240th are matched and the other characters from the 241st are unmatched are recognized as the same identifiers.
Maximum number of names	Section names : Up to 65535/file Symbol names : Up to 65535/file Module names : Up to 65535/file Note: The number may be limited by the capacity of the development environment system memory.

3.1.6 Input File Names

The C compiler identifies the type of input file by its extension and then starts the process required for that file, such as, compiling the file if it is written in C language or linking the file if it is object module. Table 3.4 shows the starting process for each type of file. (See Figure 1.1 “Programming Flow by CC32R”.)

Table 3.4 Input File and First Procedure

Extension	Type determined by the C Compiler	First Procedure
.c	C language source file	Compiling
.ms	Assembly language source file	Assembling
.mo	Object module file	Linking
Others	Object module file	Linking

3.1.7 Output File Naming

The name of the output file is the one specified by the -o option. If this option is not used, the C compiler automatically gives the name to the output file as shown in Table 3.5.

The -o option (lower case) is ignored if two more input files are specified and the output is not a load module. The output file is given the name according to Table 3.5.

Table 3.5 Output File Names (default)

File name	Description
file`.mi`	The name of the preprocessed C language source file. The file name is the name of the C language file with the extension replaced as .mi.
file`.ms`	The name of the compiled assembly language source file. The file name is the name of the C language file with the extension replaced with .ms.
file`.mo`	The name of the assembled object module file. The file name is the name of the source file with the extension replaced with .mo.
`a.mout`	The linked load module file.

3.1.8 Output During Execution

When two or more input files are specified, the compiler outputs the following status information during execution :

- Upon start of compiling :

  The name of file being compiled is output (when the input file is a source file).

- Upon start of linking :

  The message “Linking” is displayed.

The C compiler will end by doing nothing if no input file name is specified. No starting or ending messages will appear.

3.2 Command Options

3.2.1 Command Options

The functions of the C compiler command options are listed in Table 3.6.

Table 3.6 Command Options for the C Compiler (1/8)

Option	Description
`-access=`*`Access Control File`*	Specified when using the base register function. Based on the contents of the Access Control File, this option determines the code for the following two types of access (read/write): (1) Access to objects (variables and structures, etc.) assigned to the default D or B section (2) Access to objects at fixed addresses. Code is generated as a command using 16-bit register relative indirect addressing mode.
`-c` (lower case)	Performs only compiling, and generates an object module file (*file*.`mo`). This option is ignored when the -S option is used.
`-C` (upper case)	The C preprocessor does not delete any comment.
`-constr`	Allocates character string constant to C section, enabling allocation in the ROM area.
`-D` *name* `-D` *name=def*	Defines the name or constant specified by the *def* into the macro named *name*. The *name*=1 if *def* is not used.
`-e` *entrypoint*	Sets the entry point of a load module at *entrypoint* (symbol). This option is effective during linking.
`-E`	Invokes the C preprocessor only. The preprocessor output is sent to the standard output.
`-float_only`	he double type is regarded forcedly as the float type. If this option is used together with the -m32re5 option, all floating point operations can be made applicable to the FPU instruction. For more details, refer to Chapter A.5.
`-fminst`	A code is generated, using FMADD (Floating-point multiply and add operation instruction) and FMSUB (Floating-point multiply and substract operation instruction). This option is disregarded where no -m32re5 option is valid. Where this option is not specified, the FMADD and FMSUB instructions are not used. For more details, refer to Chapter A.5.
`-g`	Outputs the information (debug information) as necessary for debugging, to the object module file or the load module file.This option had been always designated. This option is always enabled.

Table 3.6 Command Options for the C Compiler (2/8)

Option	Description
`-I` *path*	Adds a *path* to the directory under which a header file is to be searched. The header file search is performed in the order shown : (1) Within the directory under which source file is stored (2) Within the directory specified by this option. (3) Within the directory for which the environment variable M32RINC is set (If not set, in the order /usr/local/M32R/include.). The search procedure (1) described above is skipped in the case of a header file search using the format : `#include <`*file*`.h>`
`-l` *lib*	Specifies a library named *lib*. The library is searched in the following order : (1) The directory specified in -L option. (2) The directory set for the environment variable M32RLIB. (If not set, /usr/local/M32R/lib.)
`-L` *dir*	Specifies the library search directory.
`-M`	Starts only the C preprocessor to identify the dependency of "makefile" and outputs the result to the standard output.
`-MAP` *map_filename*	Outputs the *map_filename* map file. This option becomes effective at linking.
`-MEM` *addr1,addr2*	This option allows writing of the C program into the ROM by assigning the sections to the appropriate memory locations. This option is a simplified version of the -SEC option and is made effective when the section is composed of P, D, B and C and cannot be used if a user made section exists. Specify the address in hexadecimal. The hexadecimal number starting with an alphabetic letter must have a 0 (zero) preceding the letter. *addr1* must be assigned the start address of the RAM area (locations for D, B section). The sections must be linked in the order of D and B. The RAM memory locations specified by *addr1* are reserved but they are not used to store the initial value data. *addr2* must be assigned the start address of the ROM area (locations for initial value data of P, C and D sections). The sections must be linked in the order of P, C and D. The D section (initial value data) is output as the section name, ROM_D. The -MEM option cannot be used together with the -SEC or -r option. Each of the following options denotes the same process : -MEM 1000,8000 -SEC @D=1000,B,P=8000,C,D

Table 3.6 Command Options for the C Compiler (3/8)

Option	Description
`-noinline`	The inline keyword is made invalid. The inline keyword,if specified, is ignored even when it is described.(This is not a error.) For more details about the inline expansion function,refer to Chapter A.4.
`-o` *output_filename* ("o" is an lower case)	Gives the name *output_filename* to the output file. If this option is skipped and if the output file is a load module file, the output file is given the name, a.mout. The *output_filename* specified by this option is also effective when the -P, -S or -c option is specified. If the number of input files is two or more, the *output_filename* is ignored and the name of output file is the input file name with extension either .mi, .ms or .mo.
`-O`*priority* ("O" is an upper case)	The option priority is used to specify a higher *priority* between code size and speed during optimization. Use the `time` priority or `space` priority by referring to the definition below. Do not separate symbols `-O` from the word *priority*. `-Otime` : Optimization with the speed has priority over code size. `-Ospace` : Optimization with the size reduction has priority over speed. `-Otime` and `-Ospace` cannot be specified simultaneously. Specifying this option without specifying `-O`*level* is equal to specifying "`-O7`" as `-O`*level*. Skipping both this option and `-O`*level* cannot start optimization. Specifying only `-O` is equal to specifying `-O`*priority* to "`-Otime`" and `-O`*level* to "`-O7`". <Examples> `-Otime` : Priority on speed, optimization at `-O7` level `-Ospace -O4` : Priority on size, optimization at `-O4` level `-Ospace -O` : Priority on size, optimization at `-O7` level `-O` : Priority on speed, optimization at `-O7` level When using this option simultaneously with the debug option (-g), the function of debugging receives an influence.For more details, refer to Section 3.2.3.

Table 3.6 Command Options for the C Compiler (4/8)

Option	Description
-O*level* (“O” is an upper case)	Specifies optimization level. Do not separate the symbols -O from the word *level* by a space character. Set -O*level* to one of the following values : 0 : No optimization 1 : Optimization of assembly language 2 : Local optimization 4 : Global optimization Note that combination of values 1, 2 and 4 achieves optimization at these levels. Specifying this option without specifying -O*priority* is equal to specifying “-Otime” as -O*priority.* Skipping both this option and -O*priority* cannot start optimization. Specifying only -O is equal to specifying -O*priority* to “-Otime” and -O*level* to “-O7”. <Examples> -O1 : Optimization at level 1, giving priority to speed -O6 : Optimization at levels 2 and 4, giving priority to speed -O -Ospace : Optimization at levels 1, 2 and 4, giving priority to size -O : Optimization at levels 1, 2 and 4, giving priority to speed When using this option simultaneously with the debug option (-g), the function of debugging receives an influence.For more details, refer to Section 3.2.3. Also, where the optimization exceeding Level 4 is effective, the inline expansion is made effective. For more details about the inline expansion function, refer to Chapter A.4.
-m32re5	A code is generated, using the M32R/ECU#5 extension instructions (FPU instructions of M32R-FPU core). Also, the non-normalized numeral in the floating point constant is reduced to "0.0". For more details, refer to Chapter A.5.
-P (upper case)	Uses the C preprocessor only and generates a file with the extension .mi (*file*.mi). The file does not include line information.
-r	Creates a load module file in the form of relocatable one. If this option is not specified, the module file is generated as an absolute file. This option becomes effective at during linking. This option cannot be used with -MEM or -SEC.

Table 3.6 Command Options for the C Compiler (5/8)

Option	Description
`-R` *old*=*new*	The C compiler changes the name of the section it is creating. The C compiler classifies the C program into the following 4 sections according to its function (default): **Section name** / **Description** P — Program C — Constant value D — Global variable having initial value B — Global variable having no initial value When changing these section names, use this option to enter the original section name (P, C, D or B) to *old* and the *new* section name to new.
`-S` (upper case)	Generates assembly language source file(s) (*file*.ms) by compiling only.

Table 3.6 Command Options for the C Compiler (6/8)

Option	Description
`-SEC` *name* `-SEC` *name=addr* `-SEC` *name=addr,name=addr...*	Specifies the linking order of sections and the start address. Enter the section name into *name*, and the address of the location for the section into *addr*. Next to the = , specify the start address in hexadecimal. Affix 0 (zero) to the first letter of the hexadecimal, if the hex. number starts with an alphabetic character. If the start address is not specified, a section is immediately followed by the next section. If the symbol "@" is affixed to the beginning of the section name, the memory reserve information for that section is output without any data (initial data elimination function of the linker). The initial data which is not output because of the elimination function of "@" can be output to another area (initial data extraction function of the linker). To output, enter the section name without "@" in the command line. The data is output under the section name, ROM_*name*. Example : `-SEC  @D=1000,B,P=0c000,C,D` This example assigns address 1000_{16} and subsequent addresses for the D section and places the D section initial data next to the C section. The initial data is output to the load module file under the section name, ROM_D. This option cannot be used with -MEM or -r.
`-switch_by_offset`	Additional designation to cc32R command line option is required at compiling. For more details, refer to Chapter 3.2.4.
`-U` *name* (upper case)	Undefines the defined preprocessor macro specified by *name*.
`-v` (lower case)	Displays the command used to invoke each subprocess of the C compiler as they are executed.
`-V` (upper case)	Outputs the invoking message to the standard error output. The other options are ignored. No processing actually takes place.
`-w` (lower case)	Disables the warning message and the information message displays.
`-warn_suppressed_nested_comment`	Generation of a warning for nested comments is suppressed.

Table 3.6 Command Options for the C Compiler (7/8)

Option	Description
`-rel16`	For access to the symbols of D and B sections located in RAM, this option outputs load/store instructions by register relative indirect addressing. Access to symbols are performed in register relative indirect addressing mode via the fixed register, R12. When using this option, pay attention to the precautions below: (1) Set the R12 register and define the "__REL_BASE" symbol. (2) If the total size of D and B sections exceed 64 Kbytes, do not use this option. (3) For const-qualified variables (located in C section) to be referenced in another source, a prototype accompanied by const must always be declared. (4) Do not destroy the content of the R12 register. Also refer to Section 3.2.2 for more information about this option.
`-XX =` *symbol_num*	This option allocates memory for symbols that are required when compiling. The required memory size is expressed by a number of symbols, which is specified in symbol_num. The default value of symbol_num is 40,000. Specify this option when compiling a source file, such as middleware, which has more than 40,000 symbols.

`-small`
`-small -memlarge`
`-medium`
`-large`
	These options specify the memory model in which to compile. If none of these options is specified, the source file is compiled in small model (-small).
	The option -small specifies that the source file be compiled for the small model (both code and data stored within the range of 0x00000000 to 0x00FFFFFF).
	The option -small -memlarge specifies that the source file be compiled for the small model with -memlarge attached (full 32-bit memory space supported for only data).
	The option -medium specifies that the source file be compiled for the medium model (code stored within the range of given address A to A + 0x00FFFFFF, and the full 32-bit memory space supported for data).
	The option -large specifies that the source file be compiled for the large model (full 32-bit memory space supported for both code and data).
	Also refer to Appendix A.2 for more information about this option.

Table 3.6 Command Options for the C Compiler (8/8)

Option	Description
`-CS`	With the -CS option, the C compiler cc32R generates an assembly language source file (default file extension ".ms") with C source file. No object file is generated.
`-stack`	Selecting the -stack option generates a stack utilization display file (a text file whose input file name extension is changed to ".stk") and generates an object file too at the same time. One stack utilization display file is generated for each source file written in C language. Stack sizes of individual functions are output to the file together with a list of function names called by them. The stack utilization display file is used as an input file to the stack size calculation utility (stk32R). Selecting this option concurrently with one of the -E, -M, and -P options generates no stack utilization display file. The use status of the stack used under the in-line assembly feature is not output. Functions defined under the in-line assembly feature are not output. Functions called under the in-line assembly feature are not output. No stack utilization display file is generated for the assembler function (the source file written assembly language). If the stack utilization display file for the assembler function is necessary, make a text file separately. For descriptions as to the stack utilization display file, see the file "stack size caluculation utility guide". For information on functions and usage of the stack size caluculation utility (stk32R), see description in the file "stack size caluculation utility guide".
`-zdiv`	For avoiding the integral zero-division problem of M32R/ECU series, to generate assembly source with inserting NOP instructions each after the all of created DIV-instructions. Also, it inserts NOP instructions as same in asm functions too. In the case of inputting assembly sources to cc32R, it performs same from assembling by as32R.
`-@`	Invoking the tool with the -@ option, you can output messages (which are output to the standard error by default) to the standard output.

3.2.2 Notes about -rel16 option to be taken when programming

When the option -rel16 is specified, the compiler outputs load/store instructions by register relative indirect addressing for access to the symbols of D and B sections located in RAM. Access to symbols are performed in register relative indirect addressing mode via the fixed register, R12.

Example: Differences in codes output with and without -rel16 option

When -rel16 option is not used	When -rel16 option is specified
LD24 R1, #_symbol LDUB R1, @R1	LDUB R1, @(_symbol-__REL_BASE, R12)

When the option -rel16, pay attention to the precautions below during programming:

- Be sure to set the R12 register and define the "__REL_BASE" symbol.

  When using this option, you need to set the R12 register and define the "__REL_BASE" symbol at the beginning of the program. Generally, these may be set in the startup program start.ms. (Refer to Section 7.3, Programming Start-up Program.") The values set for the R12 register and "__REL_BASE" symbol must be the start addresses of the D and B sections plus 32 Kbytes. For example, if linked in order of D and B sections, with the total area of 64 Kbytes and the start address of D section = h'20000, then you set the value h'28000. (See Figures 3.1 and 3.2.)

- Make sure the total size of data in D and B sections is within 64 Kbytes.

  When using this option, make sure the total size of data in D and B sections is within 64 Kbytes, and that the data are located at contiguous addresses. If the total size of data in D and B sections exceeds 64 Kbytes, do not specify this option. Otherwise, a "relocation size overflow" or another error may occur, because the 16-bit displacement is exceeded. (See Figures 3.1 and 3.2.)

- Prototype declaration of const-qualified variables.

  When const-qualified variables (located in D section) are going to be referenced in another source, a prototype accompanied by const must always be declared.

  Example: When a variable declared in program 'b' is referenced in program 'a'.

Program 'a'

```
extern          int  aa;
extern    const int  bb;   /* const */
void foo(void)
{
    if ( aa > bb )
          .....
    else
          .....
}
```

Program 'b'

```
      int  aa;
const int  bb;
void test(void)
{
    aa = 0;
    bb = 1;
    foo();
}
```

- Do not destroy the content of the R12 register.

  Because the R12 register is used, from start to end of the program, as the Rsrc register for register relative indirect addressing, be careful not to destroy the content of the R12 register.

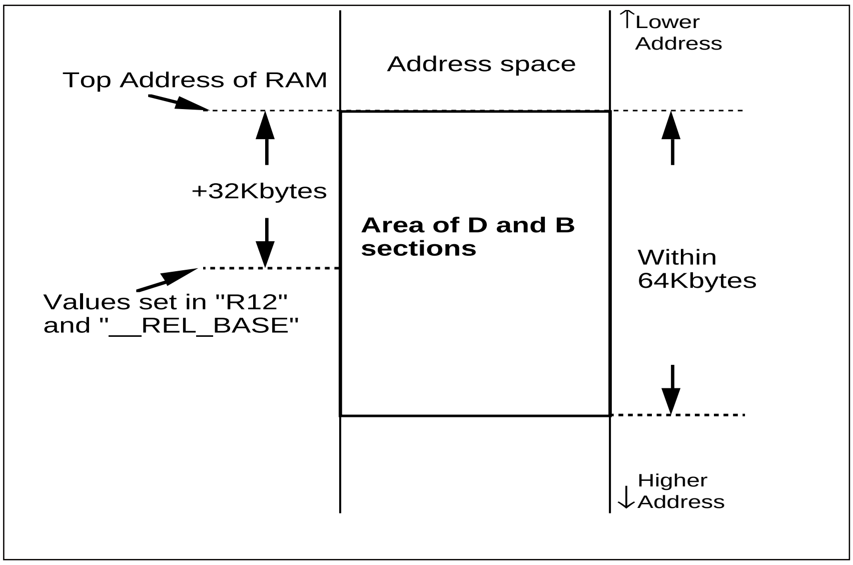


Figure 3.4 Values to be set in R12 and __REL_BASE

```
                ADDI     R6, #-1
                BGTZ     R6, loop1
loop_cntl:
                .export   __REL_BASE        ;Newly added
__REL_BASE      .EQU      h'28000           ;Newly added
                LD24     R12, #__REL_BASE   ;Newly added
                BL       $_c_main
                .END
```

Figure 3.5 Setup examples of R12 and __REL_BASE (start.ms)

3.2.3 Debugging Limitations when Optimize Options Are Specified

The compiler is always specified the function for debugging. Therefore, the compiler allows for source-level debugging during optimization.However, unnecessary lines or variables may be deleted or the sequence of evaluation may be changed as a result of optimization. Therefore, it is only when the conditions shown below are met that the values of variables can be verified by the debugger.

This limitation does not apply when optimization is not specified.

- At a breakpoint at the entry of a function, any variable from (1) to (4) shown below can be checked ("Breakpoint at the entry of a function" is the first statement in which it is possible to set a breakpoint within the function.):
  (1) A global variable
  (2) A static-declared global variable used inside that function
  (3) A static-declared local variable used inside that function
  (4) A function parameter
- At a breakpoint at the exit of a function, any variable from (1) to (3) shown below can be checked ("Breakpoint at the exit of a function" is the last statement in which it is possible to set a breakpoint within the function.):
  (1) A global variable
  (2) A static-declared global variable
  (3) A static-declared local variable used inside that function
- At the other breakpoints, any variable from (1) to (2) shown below can be checked :
  (1) A global variable which is type struct, union, or array
  (2) A local variable which is type struct, union, or array used inside that function

3.2.4 "-switch_by_offset" Option

When the compiler generates a code, using the table jump command for the switch statement, an offset table excellent in ROM efficiency is created.

[PRECAUTIONS]
The offset table generated when this option is designated can be handled only up to the range of 32K bytes with the initial case of switch statement as origin.

Accordingly, if such a large switch statement as exceeds 32K bytes is described, the code may be unable to be generated normally. (There arises overflow in linking with linker lnk32R). At this time, remove the -switch_by_offset option for re-compiling effect.

[Format:]

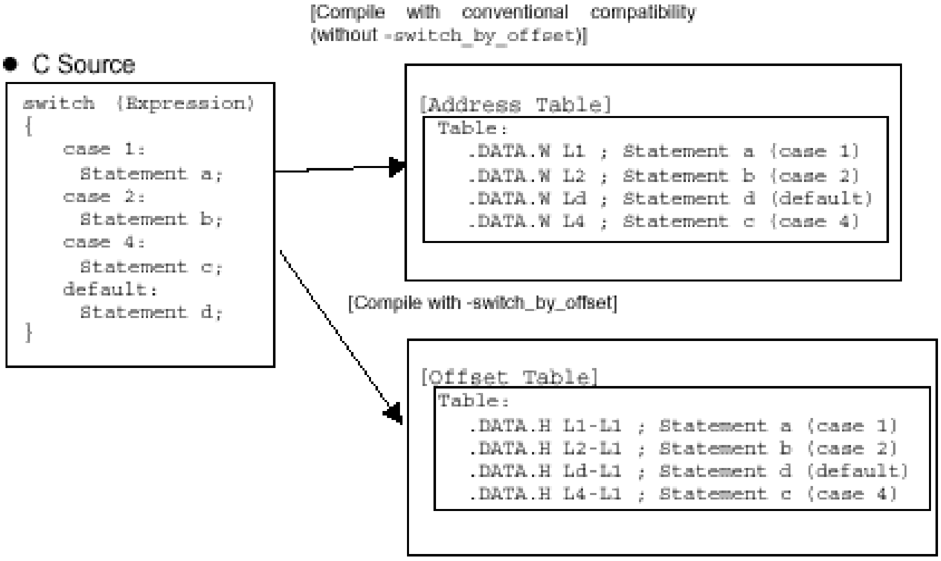
```
Example)

        cc32R [] -switch_by_offset [] filename.c
```

Additional designation to cc32R command line option is required at compiling.

[Effect:]
Where the switch statement is generated by the table jump command with the conventional CC32R, the ROM size can be reduced.

[Actually generated code:]
The following is the branch table of switch statement generated by CC32R with conventional compatibility and "-switch_by_offset".

- For explanation, only the branch table (address table & jump table) is described.
- The top label of statements a, b, c, d shall be L1, L2, L3 and Ld respectively.
- Where the ROM size becomes larger than in linear searching, the linear search is used without employing the address table and offset table, but explanation is given here on the premise that the branch table is always used.



[Compile with conventional compatibility (without -switch_by_offset)]
● C Source
[Compile with -switch_by_offset]

```
switch (Expression)
{
    case 1:
     Statement a;
    case 2:
     Statement b;
    case 4:
     Statement c;
    default:
     Statement d;
}
```

```
[Address Table]
 Table:
    .DATA.W L1 ; Statement a (case 1)
    .DATA.W L2 ; Statement b (case 2)
    .DATA.W Ld ; Statement d (default)
    .DATA.W L4 ; Statement c (case 4)
```

```
[Offset Table]
Table:
    .DATA.H L1-L1 ; Statement a (case 1)
    .DATA.H L2-L1 ; Statement b (case 2)
    .DATA.H Ld-L1 ; Statement d (default)
    .DATA.H L4-L1 ; Statement c (case 4)
```

[Explanation]

The address table generated with conventional compatibility indicates the array with which the branch addresses are stored. Each element of the array is 32 bits (4 bytes).The switch statement is branched to the addresses shown by Table [Expression - 1] after checking to see if the expression value is 1 to 4. If "-switch_by_offset" is provided at compiling, the address table becomes the offset table.

The offset table also shows the array, but each element becomes offset (distance from L1) in place of the branch address, and each element of the array is as small as 16 bits (2 bytes). The compiler generates a code branched to the address with the origin (L1) added to the offset value shown in Table [Expression - 1] for the switch statement.

3.3 Command Line Example

An invocation example of the C compiler is shown here (% is a prompt, <RET> shows an inputting return key) :

-
  ```
  % cc32R -c -g -v test0.ms test1.c test2.c<RET>
  ```

  The -c option generates the object module files for test0.ms, test1.c and test2.c (*file*.ms is an assembly language source file and *file*.c is a C language source file). The name of each object module file is the name of the source file with the extension changed to .mo (test0.mo, test1.mo, test2.mo).

  With -g, debug information is included in each object module generated by the compiler.

  The -v option allows verification on the screen, at the start of each phase of the C compiler.

3.4 The other notes

- Stack frame capacity limit (limit to size of auto variable)

The maximum stack frame that can be assigned per function is 32,764 bytes. An error results if you attempt to secure more stack frame, and no code is generated. Data over 32,764 bytes should be secured as static or global data.

Code example that an error arises :

```
void foo(void)
{
        char   x[32765];    /* Over 32,764 byts */
        x[0] = 0;
}
```

- Use of the run-time library for assigning or returning struct

In your program, if assignment of a structure or a function which returns a structure as a return value is written, an error may arises at link-time. (The linker error message "`error: external symbol not defined: $_100_builtin_memcopy`" is displayed to show that the $_100_builtin_memcopy function does not exist.) This occurs because the assignment and setting of the return value are performed by the run-time library function $_100_builtin_memcopy .

The $_100_builtin_memcopy function is included in the ANSI-C standard library (m32RcR.lib,etc ...) for stack-passing parameters, or m32RcR.lib,m32RcRM.lib,m32RcRL.lib for register-passing parameters). Specify either one of these libraries when linking. Code example that an error arises at link-time :

```
struct s {
    char xx[100];
   }x,y;

    void foo(void)
    {
          x = y;   /* assignment of a structure */
    }
```

- Notes on data access

The C compiler generates the code by selecting the optimum instruction for the size of the data type being accessed. Therefore, if your program attempts to access data that is not aligned via the pointer, an address exception (AE) arises at run-time. This may also occur with the type of casting shown in the example below.

```
Code example in which an exception may occur :
long *p;
char array[10];   /* mapped to 4-byte boundary address */

void foo(void)
{
        p = (long *)&array[1];  /* address not on 4-byte
                                   boundary assigned as long* */
        *p = 1;   /*  access using the ST instruction, but as
                      address is not on 4-byte boundary,
                      an address exception (AE) occurs */
}
```

- Notes on setting the stack pointer

  For loading and storing data on the stack, the C compiler outputs code for the LD, ST, LDH, and STH instructions, etc. Therefore, when you set a value into the stack pointer in the startup program or in a user program, be sure to specify an address that is on a 4-byte boundary. (See the sample startup program in the “7.3 Programming the Start-up Program”.)If you specify an address that is not on a 4-byte boundary, an address exception (AE) may occur when the program is run.

- Calling the floating-point operation function

  If you compile a program written in C language that performs a floating-point operation, there can be instances in which a code is internally generated that calls the floating-point operation function (_100_F~). The floating-point operation function is included in the library files (m32RcR.lib, etc...). With the code for calling the floating-point operation function generated, an error results if you don't specify the library files in performing linker. For this reason, specify the library files in performing linker even for a program that doesn't use the C's standard library functions.

- The problem that a module name that starts with a numeric turns to "ASM32R_MPRO"

  Don't generate an object file whose name starts with a numeric. If you generate an object file whose name starts with a numeric, the module name becomes "ASM32R_MPRO", and module names duplicate when generating a library, as a result, library files cannot be generated. Also, the module name within a map file generated by use of map32R becomes "ASM32R_MPRO".

  Example:

  ```
  (1) cc32R  -o  1234.mo  file.c
  (2) as32R  -o  1234.mo  file.ms
  ```

- Notes about MS-Windows(PC) version

  (1) Floating-point numbers

  For the floating-point value -0.0, the code for +0.0 are output

  (2) Path delimiter

  The path delimiter symbol is the backslash (\).
  However, to specify the path for included files specified in the source file, you can use either \ or /. In this case, / is only recognized when input and is internally replaced by \.

Therefore, the path delimiter output into warning/error messages, listing files, and debugging information is always \.

(3) Upper/lower case in a file name

A file name is upper/lower case insensitive .

For example, “file.c”, “FILE.C” and “FiLe.C” are all process -ed as the same file name.

However, upper/lower case in the generated file is effective for the long file names.

(4) Specifying Path

You do not use the relative path name with a drive letter.

Chapter 4

C Programming Language Specification

This chapter describes basic specifications of the ANSI-C programming language processed by the compiler (e.g., the elements of the C programming language and the structure of a C language source program).

4.1 Token

A token is the minimal lexical element of the C language text for manipulation and analysis by the compiler. The compiler manipulates and analyzes the following elements as tokens in conformance with the ANSI Standard :

- Keywords (see 4.1.1)
- Identifiers (see 4.1.2)
- Constants (see 4.1.3)
- String Literals (see 4.1.4)
- Operators (see 4.1.5)
- Punctuators (see 4.1.6)
- Comment (see 4.1.7)

4.1.1 Keywords

A keyword is used as a reserved word in the C language. In translation phases of the compiler, the words in Figure 4.1 are keywords.

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

Figure 4.1 Keywords

4.1.2 Identifiers

Identifiers are names listed as follows :

- Function name, Variable name
- Label name
- Tag name of a structure, union or enumeration
- Member name of a structure, union or enumeration
- Macro name
- Object name
- Typedef name

An identifier must begin with either a letter (Table 4.1) or an underscore (_) . The other part can contain letters, digits (Table 4.1), and underscores. The first 240 characters are significant for an identifier.

Table 4.1 Letters and Digits

Called	Character(s)
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 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
Digits	0 1 2 3 4 5 6 7 8 9
Alphanumerics	Both letters and digits

A region of program text in which identifiers can be used (called “scope”) is shown below (Table 4.2) :

Table 4.2 Scopes of Identifiers

Identifier	Scope
• A variable declared outside a function • A function prototype declaration • A function definition	Each of the identifiers has "file scope" which begins at its declaration or definition and terminates at the end of the file.
A parameter in a function definition	The identifier has "block scope" which begins at the { and ends at the } .
A parameter in a function prototype declaration	The identifier has "function prototype scope". It is visible only in the function prototype declaration in which it appears.
A label	The identifier has "function scope" which begins at its declaration or definition and terminates at the end of the function.
A variable declared inside a block	The identifier has "block scope". If the identifier is the same as an identifier outside the current block, in the current block, the external identifier is invalid. The external identifier becomes visible after the current block terminates.

There are four kinds of name spaces of identifiers (see Table 4.3) :

Table 4.3 Name Spaces of Identifiers

Name Space	Identifier in the Name Space
Function Name, Variable Name	An identifier which is not any of the identifiers, a label name, a tag name, and a member name.
Label Name	An identifier which is followed by a colon (:) and is in goto statement.
Tag Name	An identifier disambiguated by following any of the keywords, struct, union, or enum.
Member Name	An identifier disambiguated by the type of the expression used to access the member via a period(.) or the operator (->).

Identifiers having different name space can have the same name because the C compiler differentiates between them as described above. The identifiers in the same space cannot have the same name. For example, the name of a label can be the same as that of a function. On the contrary, a structure tag and union tag cannot have the same name .

||||| Note |||||

A key word cannot be used as an identifier (see 4.1.1).

4.1.3 Constants

A constant represents a constant numerical value. It has a type determined by its form and value, as follows :

• Floating-point constants	(See 4.1.3.1.)
• Integer constants	(See 4.1.3.2.)
• Enumeration constants	(See 4.1.3.3.)
• Character constants	(See 4.1.3.4.)

4.1.3.1 Floating-Point Constant

A floating-point constant describes a floating-point number. It consists of a mantissa (a fractional constant or a digit sequence) , an exponent part, and a suffix as shown follows :

Example :

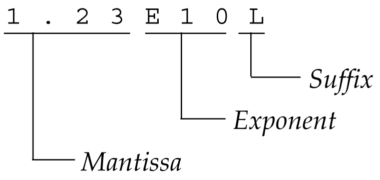


Table 4.4 Floating-Point Constant Representation

Part	Format	Description
Mantissa	*fractional_constant*	A numeric character string containing a floating-point constant and a decimal point. It can be one of the following types : *integer . fraction* Example : `3.12` *integer .* Example : `123.` *.fraction* Example : `.12`
	digit_sequence	Decimal point is not included. Example : `123`
Exponent	`e` [+ | -] *exponent* or `E` [+ | -] *exponent*	A description to express a constant as the base 10 exponent. Sign (-, +) can be omitted. The exponent is decimal number.

Suffix	`L`, `l`
	`F`, `f`
	Without
		Indicates a long double type constant
		Indicates a float type constant
		A double type constant

Note) [ ] encloses optional text. | divides choices (Select one).

If the mantissa is a *fractional_constant* (a digit sequence containing a floating-point) , it is clear that the number is a floating-point constant and the exponent part can be omitted. If the mantissa consists of only *digit_sequence*, the number is either a floating-point constant or a different constant, and the exponent part is required if it is a floating-point constant.

- Expression when the mantissa is *fractional_constant* :

fractional_constant [*Exponent*] [*Suffix*]

- Expression when the mantissa is numerical character string :

digit_sequence *Exponent* [*Suffix*]

Note) The content in [ ] can be omitted.

The examples in Table 4.5 show floating-point constant expressions.

Table 4.5 Expression of Floating-Point Constant

Example	Description
`1.0`	When the mantissa is a floating-point constant (including a decimal point), the exponent part can be omitted.
`10E3`	When the mantissa is a numerical character string (not including a decimal point), the exponent part cannot be omitted.
`1.0E1L`	If an L or l is suffixed to the end of the data, it becomes a long double type data.
`1.0E1f`	If F or f is suffixed to the end of the data, it becomes float type data.
`1.0E1`	If F or f or L or l is not used, the data becomes double type data.
`.1E1`	The number can start with a decimal point.

4.1.3.2 Integer Constant

The integer constant is used to express integer. It starts with a number and does not include exponent and fractional parts. The following three radix numeration systems can be used :

Table 4.6 Expression of Integer Constant

Radix	Consists of Base and Integer Constant.
Decimal number	Starts with a number other than 0 followed by numbers. Numbers : 0 1 2 3 4 5 6 7 8 9
Hexadecimal number	A hexadecimal number starting with 0x or 0X. Hexadecimal numbers : 0 1 2 3 4 5 6 7 8 9 a b c d e f or A B C D E F
Octal number	An octal number starting with 0. Octal numbers : 0 1 2 3 4 5 6 7

When the character u or U is suffixed to an integer constant, the constant is treated as an unsigned constant. If neither character is suffixed, the constant is treated as a signed constant. When the character l or L is suffixed to an integer constant, the constant is treated as a long type constant. If neither character is suffixed, the constant is treated as an int type constant.

Table 4.7 Integer Type Constant Data Type

Following constant	Process by C compiler (data type)
u or U suffixed	Unsigned constant
No u or U suffix	Signed constant
l or L suffixed	long type constant
No l or L suffix	int type constant

Table 4.8 shows examples of integer type constant expression.

Table 4.8 Description of Integer Type Constant

Example	Description
`123`	Signed int type decimal number 123
`123u`	Unsigned int type decimal number 123
`123l`	Signed long type decimal number 123
`123ul`	Unsigned long type decimal number 123
`0123`	Signed octal number 123
`0x123`	Signed hex. number 123

4.1.3.3 Enumeration Constant

The enumeration constant is a member of type enum and has type int.

Example : `enum rgb{ red, green, blue }`

The red, green and blue are enumeration constant.

4.1.3.4 Character Constant

The character constant represents the character or escape sequence and is enclosed by single quotation marks. To include a quotation mark (') itself in a character constant, prefix the mark \' to " ' " (\'). The following escape sequences can be used.

```
\'   \"   \\   \?   \a   \b   \f   \n   \r   \t   \v
```

Figure 4.2 Escape Sequence (Character Constant)

To express a hexadecimal number prefix \x, to express an octal number, \ . Valid hex. and octal number consists of up to 3 figures.

Table 4.9 describes how to write character constant.

Table 4.9 Example of Character Constant Expression

Example	Description
'\' '	Escape sequence. Represents single quote.
'\n'	Escape sequence. Represents line feed.
'\0'	Octal number. Represents character code 00 (hex).
'\x7'	Hexadecimal number. Represents character code 07 (hex).
'\xFF'	Hexadecimal number. Represents character code FF (hex).

4.1.4 String Literals

Enclose a string literal with double quotations ("). The escape sequences shown in Figure 4.3 can be used in a string literal.

```
\'   \"   \\   \?   \a   \b   \f   \n   \r   \t   \v
```

Figure 4.3 Escape Sequence (String Literal)

To express a hexadecimal number, use the prefix \x; to express an octal number, use \.

4.1.5 Operators

An operator performs an operation. Figure 4.4 shows the operators that are available.

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

Figure 4.4 Operators

"[" and "]", "(" and ")", "?" and ":" are used as a pair (an expression may be sandwiched between them.)

Table 4.10 describes the operators and their functions.

Table 4.10 Operators (1/2)

Operator	Description
`[]`	Array reference
`()`	Casting (e.g., (int))
`.`	Structure/union reference
`->`	Structure/union pointer reference
`++`	Pre-increment/post-increment
`--`	Pre-decrement/post-decrement
`&`	Bitwise AND Address of (as an unary operator)
`*`	Binary Multiply Pointer dereference (as an unary operator)
`+`	Binary Add Unary add (as an unary operator)
`-`	Binary Subtract Unary Negate (as an unary operator)
`~`	Bitwise complement

Table 4.10 Operators (2/2)

Operator	Description
!	Logical Negate
sizeof	Returns the operand size in bytes
/	Divide
%	Modulo
<<	Left shift in bytes
>>	Right shift in bytes
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equal
!=	Not equal
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
?:	Conditional expression
=	Assignment
*=	Multiply and assign
/=	Divide and assign
%=	Take modulo and assign
+=	Add and assign
-=	Subtract and assign
<<=	Shift left and assign
>>=	Shift right and assign
&=	Bitwise AND and assign
^=	Bitwise XOR and assign
|=	Bitwise OR and assign
,	Evaluate from left of a comma to right and assume the rightmost value.

4.1.6 Punctuators

Punctuators direct other tokens or specify a range. The punctuators in Table 4.11 are available.

Table 4.11 Punctuators

Punctuators	Applications
[ ]	Declaration of array.
( )	Declaration of function or variable.
{ }	Member declaration of structure or union. Punctuation of initial value.
*	Declaration of pointer.
,	Punctuation of initial value. Punctuation of parameter of function.
:	Declaration of bit field. Punctuation of label.
=	Start of initial value.
;	End of declaration or statement.
...	Declaration of prototype of function (used when the number of parameters is variable).

“[“ and “]” , “(“ and “)”, “{” and “}” are used as a pair (An expression may be sandwiched between them.).
A punctuation character can be recognized as an operator depending on how it is written.

4.1.7 Comment

A comment is text, though embedded in a program,that is not processed by the compiler.It starts with /* and ends with */.A comment cannot be nested.

To describe a comment in Japanese, Use the character code that designated it to M32RKIN environment variable. However, when the M32RKIN environment variable is undefined, use EUC (extended UNIX code) for EWS-version CC32R; or use shifted JIS code for MS-Windows(PC) version CC32R.

The compiler also allows C++ like comments to be written in the source program. A comment can start from "//."

```
Example:
void foo(void)
{
     char   x[3];
     x[0] = 0;  // Comments can be written in this manner

}
```

4.2 Data Types

4.2.1 Types and Type Specifiers

The type (data type) determines the meaning of a value stored in an object or a return value from a function. The types supported by the C compiler and their type specifiers (identifiers in C) for declaration are:

- **Basic types**

Character types	`char` (=`signed char`), `unsigned char`
Signed integer types	`char` (=`signed char`), `int` (=`signed int`), `short` (=`signed short`, `signed short int`), `long` (=`signed long`, `signed long int`)
Unsigned integer types	`unsigned char`, `unsigned int`, `unsigned short` (=`unsigned short int`), `unsigned long` (=`unsigned long int`)
Floating types	`float`, `double`, `long double`

- **The others**

Array types	
Structure types	`struct`
Union types	`union`
Enumeration types	`enum`
Void type	`void`
Pointer types	
Function types	

`signed` and `unsigned` are type specifiers that indicate whether an entity is either signed or unsigned. As for an array, pointer, or function assigned no type specifier, you specify that either an object or a function is of the type you declare in a fixed format in ANSI-C.

As for a type specifier that is alternatively represented by (= *alias*) such as (= `signed char`), the C compiler translates the alternative representation as having the same meaning. In ANSI-C, the definition of signed or unsigned integer type includes the character types (see Table 4.12), so both `char` and `unsigned char` are given in the above list.

||||| Note |||||

A char type data without signed or unsigned specification is recognized as a signed char data.
An int type data without signed or unsigned specification is recognized as a signed int data.
A long int type data without signed or unsigned specification is recognized as a signed long data.
A short int type data without signed or unsigned specification is recognized as a signed short data.
A enum type data is a signed int type data.

4.2.2 Types

Types are classified into several groups according to their properties or standpoints. For example, the integer types (including the character types), the floating types, and the pointer type are generically called scalar types. Such groups are shown in Table 4.12 and in Figure 4.5, and other representations of types are shown in Table 4.13. This manual uses these terms to give explanations in some instances.

Table 4.12 Types

Called	Types
Character types	`char`, `unsigned char`
Signed integer types	`char`, `int`, `short`, `long`
Unsigned integer types	`unsigned char`, `unsigned int`, `unsigned short`, `unsigned long`
Floating types	`float`, `double`, `long double`
Basic types	signed integer types, unsigned integer types, floating types
integer types	signed integer types, unsigned integer types, enumerated types (`enum`)
Arithmetic types	integer types, floating types
Scalar types	integer types, floating types, pointer types
Aggregate types	structure types (`struct`), array types
Derived declarator types	array types, pointer types, function types
Derived types	array types, pointer types, function types, structure types (`struct`), union types (`union`)

Table 4.13 Type Representations

Called	Description
Composite types	They are constructed from two types which are compatible.
Incomplete types	They describe object but lack information needed to determine the sizes of objects (e.g., an array type unspecified its size).
Object types	Describe objects without function types.
Function types	Describe functions. They determine types of the return value and parameters, and the number of parameters. **Example :** `char *(*func1)(int,int)` `func1` is the pointer to a function which returns "a pointer to char" and has two `int` type parameters.
Qualified types	Types with any type qualifier (`const` or `volatile`).
Unqualified types	Types without any type qualifier (`const` or `volatile`).
Top type	In a derived type, the top type is outermost derivation represented by type specifiers and qualifiers. In a no derived type, the top type is itself. **Example :** `const int *i;` This means that `i` is "pointer to qualified int". The top type is "pointer type" and is not "qualified type" or "int type".

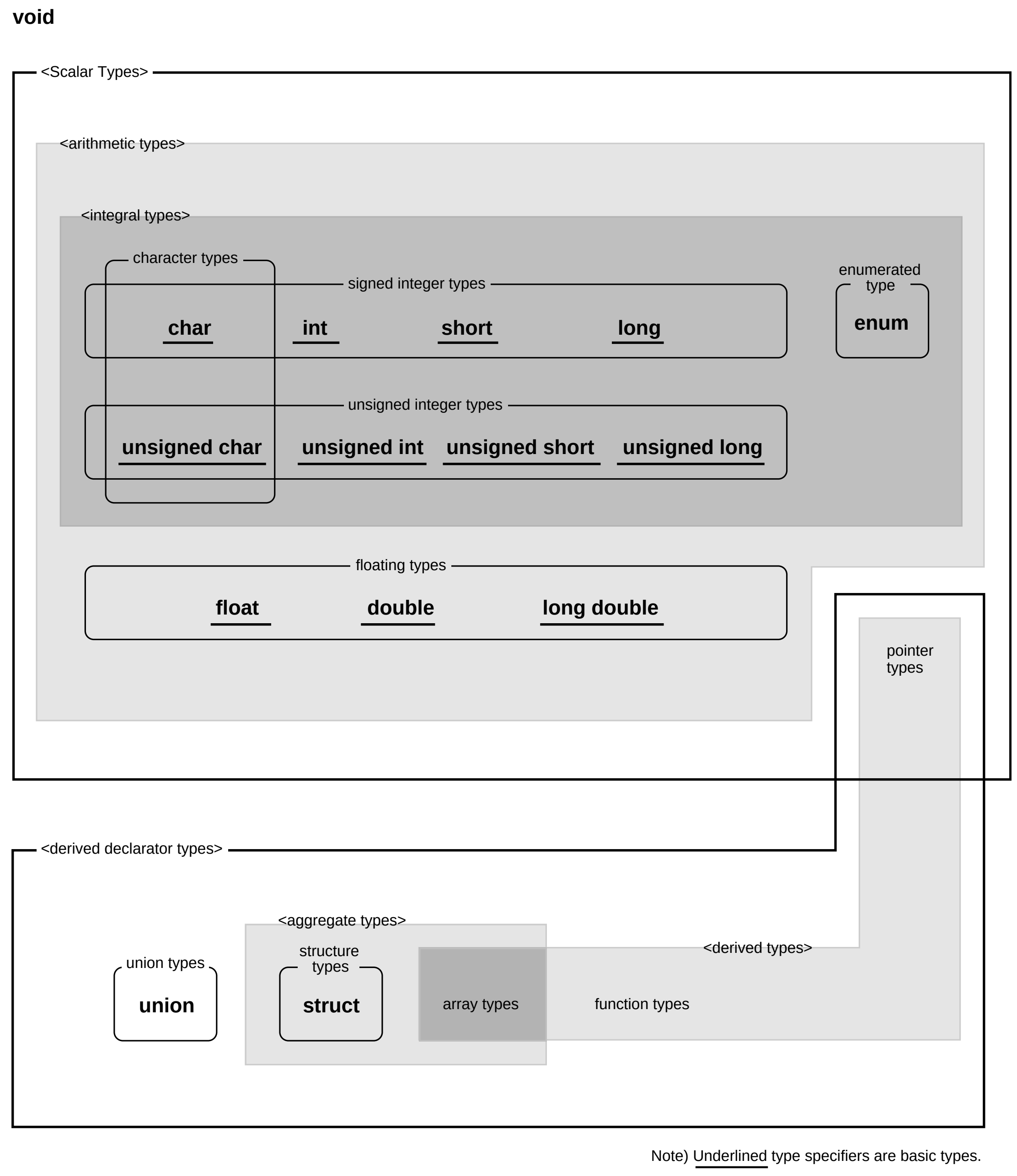


Figure 4.5 Types

4.2.3 Data Size and Range of Basic Types

The size and limits (maximum and minimum values that can be expressed) of basic type data are as shown in Table 4.14. For further information, see Chapter 5 "Internal Data Representation".

Table 4.14 Data Type and Size

Data Type	Size	Minimum Value	Maximum Value
char unsigned char	1-byte(8-bits)	-128 0	127 255
short unsigned short	2-byte(16-bits)	-32768 0	32767 65535
int unsigned int	4-byte(32-bits)	-2147483648 0	2147483647 4294967295
long unsigned long	4-byte(32-bits)	-2147483648 0	2147483647 4294967295
float	4-byte(32-bits)	1.17549435e-38F	3.40282347e+38F
double long double	8-byte(64-bits)	2.2250738585072014e-308	1.7976931348623157e+308

4.2.4 Data Format for Floating-Point Constants

Floating type data is assumed to be in the IEEE-754 format. IEEE-754 is the internal representation of real numbers in the form specified by Institute of Electrical and Electronics Engineers (IEEE). For internal representation of floating type data, see Chapter 5, "Internal Data Representation".

4.2.5 Type Qualifiers

Type qualifiers (const and volatile) can be added to any data type supported by the C compiler. The type qualifiers are :

- const

  The type qualifier indicating that the value of the object cannot be changed. The value of an object, once declared by a data type such as const, cannot be replaced.

  Example : `const char c=1;` ——— (a)
  `const char *ptr;` ——— (b)

  (a) Variable c is fixed at 1 and cannot be replaced.
  (b) The content specified by the pointer ptr is fixed and cannot be changed. The ptr itself can be changed.

- volatile

  The type qualifier implying that the object can be changed. Once volatile is specified, the C compiler will not optimize the object and directly outputs writing and reading processes to the object code.

  For example, specifying volatile to an area or a memory map I/O area suggests that the contents in that area may be changed.

  In the following example, the C compiler does not perform optimization and outputs objects to the object codes.

```
volatile  int i;
volatile  int j;
                 j = i;
                 j = i;
```

4.2.6 Storage Class Specifiers

Compiler-supported types can be prefixed the following 5 storage class specifiers. If a variable that appears inside a function body is not specified a storage class, `auto` is assumed.

- auto — Declares that an object under consideration is of automatic storage class. You can make this declaration only on an object within a function, and the object turns effective within a block only (the part between { and }). When the block finishes, the storage area is released.

  Example : `auto int a=1;`
  `a` is visible only inside the block in which this declaration appears.

- extern — The object or function declared "extern" is the external storage class. The extern declaration is to make an object or function with global declaration visible in the current source file.

  Example : `extern int val;`
  The declaration to use in the source file in which this declaration appears the variable `val` declared in another source file.

- static — The object or function declared "static" is the static storage class. It is visible only in the source file in which its static declaration appears.

  Example : `static void f(int);`
  The prototype declaration to use only in the source file in which this declaration appears the function `f` declared in the same source file.

- register — The object declared "register" is the register storage class. The register declaration requires the C compiler that an object that will be accessed frequently can be faster accessed. "register" can be declared for only an object inside a function. The object declared "register" has block scope (visible only between { and }). Such object is called "register variable".

  Example : `register reg;`
  The variable reg is used frequently.

- typedef By using “typedef”, you can define any name for a type. (In the ANSI standard, “typedef” is a storage class specifier for convenience. However, no storage is created.)

  Example : `typedef char *   STR;`
  `STR` can be used as a type specifier. The type of an object declared by using `STR` is a pointer to char. (i.e., `STR str;` equals `char *str;` .)

4.3 Conversions

4.3.1 Explicit Conversions (Cast)

You can temporarily convert (cast) the type name of an object or a function by use of the cast operator.

The syntax for a cast is show in Figure 4.6. A cast to an lvalue is not allowed.

Syntax

(*new_type_specifier*) *identifier*

where:

new_type_specifier	: Either a scalar type or void.
identifier	: An identifier declared as a scalar type.

Example

```
char *str = "abc";
int *a = (int *)str;
```

In order to substitute the operation value of int type a for double type x, it convert into a double type.

Figure 4.6 Syntax of Explicit Conversion (Cast)

Although a type is converted to its compatible type, the value in the object or from the function is unchanged. However, making conversion to an incompatible type changes the value as shown in Table 4.15 and 4.16 in 4.3.2 “Implicit Conversion”.

4.3.2 Implicit Conversions

Even though no an explicit conversion is specified, a conversion may be done by the C compiler. This is termed "implicit conversion", and includes the following :

- Conversion of enumerations

  A constant declared as `enum` is assumed as `int`.

- Conversion of characters and integers

  If an `int` can be represented all values of the original type, the value is converted to an `int`. Otherwise, it is converted to an `unsigned int`. This implicit conversions termed "integral promotion".

- Conversion of signed/unsigned integers

  Results of conversions from a signed integral type to an unsigned integral type or vice versa are as shown in Table 4.15.

Table 4.15 Signed/unsigned integers Conversions

Original	New Type (after conversion)	Value after Conversion
Positive signed integer	Unsigned integer has equal or greater size	Unchanged.
Negative signed integer	Unsigned integer has equal size	"The original value + (the maximum value for unsigned type +1)"
	Unsigned integer has greater size	First, the original value is promoted to the signed integer corresponding to the unsigned integer. Then, the value is converted to unsigned by : "the result in promotion + (the maximum value for the unsigned integer type +1)".
Unsigned integer	Signed integer has equal size	The low bits in the original value will be copied to the shorter signed integer. (The highest bit means a sign).
Signed/unsigned integer	Signed integer has shorter size	The low bits in the original value will be copied to the shorter signed integer. (The highest bit means a sign).
	Unsigned integer has shorter size	The positive remainder on : "The original value ÷ (1 + the maximum value for the new type)"

- Floating and integral

Results of conversions from a floating type to an integral type, vice versa and from a floating type to another floating type are as shown in Table 4.16.

Table 4.16 Floating and Integral Conversions

Original	New	The Value after Conversion
Integral type	Floating type	It is unchanged if the value can be represented with the new type. It is rounded to the nearest value which can be represented with the new type if the value cannot be represented exactly.
Floating type	Integral type	The fractional part is discarded.
Floating type	Shorter floating type	It is unchanged if the value can be represented with the new type. It is rounded to the nearest value which can be represented with the new type if the value cannot be represented exactly.
float	double or long double	Unchanged.
double	long double	Unchanged.

- Usual arithmetic conversions

In performing a binary operation, the types of the operands are converted implicitly in order to yield a common type (as shown in Table 4.17) , which is also the type of the operation result. These are termed “usual arithmetic conversions”.

Table 4.17 Usual Arithmetic Conversions

Either operand	The other operand	Usual arithmetic conversion
long double	non long double	The other operand is converted to long double.
double	non double	The other operand is converted to double.
float	non float	The other operand is converted to float.
unsigned long	non unsigned long	The other operand is converted to unsigned long.
long	unsigned int	If the other operand is represented in long, it is converted to long. Otherwise, both operands are converted to unsigned long.
long	non long	The other operand is converted to long.
unsigned int	non unsigned int	The other operand is converted to unsigned int.

Otherwise in this table, both operands have type int.

||||| Note |||||

Be warned that implicit conversions are yield, when you write a program.
For example, as shown in the example given below, when some integer which has type unsigned int (its value will be equal to or greater than 0) is compared to the data -1 which is assumed int, the comparison result is that the integer is greater than the -1 in logical, however a converse judgement is actually made.

```
unsigned int u;

if( u>(-1) ){              /* expected relation yields true in logical */
     printf("True");       /* if true, this printf is executed */
}else{
   printf("false");        /* in actual, the relation yields false and
                           this printf is executed */
}
```

This is because following implicit conversions are executed.

(1) According to usual arithmetic conversions, the -1 (type int) is converted to type unsigned int.

(2) According to signed/unsigned integers conversions, when (1), the -1 is converted to 0xffffffff calculated from “the original value + (the maximum number for unsigned type +1)” shown in Table 4.15 , that is,
“-1 + 0xffffffff, which is the maximum value for int +1”.

4.4 Preprocessing Directives

A preprocessing directive starts with # and processed by the C compiler in the C preprocessor phase. Preprocessing directives include the commands shown in Table 4.18.

Table 4.18 Preprocessing Directives

Preprocessing Directive	Description
#include	Insert file
#define	Defines macro
#undef	Undefines macro
#if	Executes conditional compile
#else	Executes conditional compile
#endif	Ends conditional compile
#elif	Executes conditional compile
#ifdef	Executes conditional compile
#ifndef	Executes conditional compile
#line	Specifies line
#error	Issues an error message and pauses process

The macros shown in Table 4.19 are predefined (reserved) .

Table 4.19 Predefined Macros

Predefined Macro	Description
_ _LINE_ _	The line number of file being compiled
_ _FILE_ _	The name of the file being compiled
_ _STDC_ _	1 when ANSI compatible
_ _DATE_ _	Current compiling date
_ _TIME_ _	Current compiling time
_ _M32R_ _	The target microprocessor is M32R

The operators shown in Figure 4.7 can be used in a constant expression.

```
(         )        &        *        +        -        ~        !
sizeof    /        %        <<       >>       <        >        <=
>=        ==       !=       ^        |        &&       ||       ?
:
```

Figure 4.7 Operators that can be used in a constant expression

4.5 System Reserved Names

The names in Figure 4.6 are reserved during development environment of the M32R and can be used in a C program.

Syntax _*Number*_*Name*

Number : 3-digit decimal number

Name : Character string consisting of alphanumerics and underscore(s) (see 4.1.2.)

_ : An underscore

Example

`_900_main`, `_900_INITLIB`

Figure 4.8 System reserved names

Reserved names are classified into the three groups. Of these names, the names numbered 000-099 and 901-999 can be used in most programs. The 3 groups are:

- System names reserved for operating systems (000-099)

  System names reserved for variables and functions offered by different operating systems (OS) and therefore, may vary from OS to OS. These names are used to interface with a particular OS.

- System names reserved for languages and libraries (100-900)

  These names are used by development support tools such as languages and libraries. Do not use these names in a program because they may be used by language processing and libraries. These names are used in the C standard libraries.

- Names reserved for user systems (901-999)

  These symbols can be used in the user program. By providing layers in the user system, symbols can be used without confliction.

The names in the object program that correspond to external definitions and reference symbols in a C program are the names used in the C program but have a underscore (_) or dollar mark ($) prefixed to the name. For examples, the system reserved names shown in Figure 4.6 are expressed as follows when used in an object file : _ _900_main, _ _900_INITLIB or $_900_main, $_900_INITLIB.

4.6 Limitations for C Language

The C compiler places various limits on coding as shown in Table 4.15.

Table 4.20 Limitations on C Language Coding (1/2)

Description	Items	Limits
Preprocessing directives	Nest level by #include statement	Practically not limited.
	Number of macro identifiers	Practically not limited.
	Number of parameters that can be specified by macro definitions and macro call	Practically not limited.
	Nesting level of #if, #ifdef, #ifndef, #else and #elif statements	Practically not limited.
	Total number of operators and operands that can be specified by #if and #elif statements	Practically not limited.
Declaration	Number of external identifiers	Practically not limited.
	Valid Number of identifiers in a block	Practically not limited.
	Number of qualifying pointers, arrays and function declarators in a declaration	Practically not limited.
	Valid Number of characters in external and internal identifiers and macro names	Up to 31 characters

Table 4.20 Limitations on C Language Coding (2/2)

Description	Items	Limits
Statements and expressions	Nest levels of compound statements, iteration structure, or selection statements.	Practically not limited.
	Number of case labels in a switch statement	Practically not limited.
	Number of parameters in a function definition or function call	Practically not limited.
Others	Size of the stack frame that can be allocated per function	Practically not limited.
	Number of nests in parentheses	Practically not limited.
	Number of initial values that can be defined by a variable definition with initialization expression	Practically not limited.
	Length of source program line	Up to 512 characters including line feed code

Chapter 5

Internal Data Representation

This chapter describes memory allocation of C program data for various types.

5.1 Data Representation on Memory

Representation of C language data on memory is determined by the following items:

- Data size — Memory size occupied by data
- Function of data — Relationship between data contents on memory (bit pattern) and C language data value. Defines the range of value of scalar type data and allocation of aggregate type data elements.
- Alignment of data (adjusting boundaries)

  If a data item is put in a location having an address of a multiple of an integer, either 2 or 4, there can be instances in which you can access it quickly, though this depends on the type of data. This integer is termed “alignment”. The C compiler allocates data according to the alignment in compliance with its type.

  A data item whose alignment is an integer *n* is dealt with as “*n*-byte alignment”. In this instance, an address of a multiple of *n* is called either “alignment boundary” or “*n*-byte boundary”. An *n*-byte alignment data item is not allocated across its alignment boundaries (*n*-byte boundaries).

5.2 Integral Types

Table 5.1 lists integral types used in C language.

Table 5.1 Internal Representation of Integral Types

Type	Size	Alignment	Signed?	Minimum Value	Maximum Value
char	1byte	1byte	Yes	-2^{7}(-128)	2^{7}-1(127)
unsigned char	1byte	1byte	No	0	2^{8}-1(255)
short	2bytes	2bytes	Yes	-2^{15}(-32768)	2^{15}-1(32767)
unsigned short	2bytes	2bytes	No	0	2^{16}-1(65535)
int	4bytes	4bytes	Yes	-2^{31}(-2147483648)	2^{31}-1(2147483647)
unsigned int	4bytes	4bytes	No	0	2^{32}-1(4294967295)
long	4bytes	4bytes	Yes	-2^{31}(-2147483648)	2^{31}-1(2147483647)
unsigned long	4bytes	4bytes	No	0	2^{32}-1(4294967295)

The MSB[Note)] of a non unsigned data indicates the sign (0 , positive data or 0, 1, negative data.). Negative data is expressed as two's complement.

An unsigned type data has no sign and its value is positive or 0.

Examples of the internal representation are shown in Table 5.2.

Note) MSB = most significant bit

Table 5.2 Examples of Internal Representation of Integers

Type	Internal Representation	Value
char	0000 0001	1
	1111 1111	-1
unsigned char	0000 0001	1
	1111 1111	255
short	0000 0000 0000 0001	1
	1111 1111 1111 1111	-1
unsigned short	0000 0000 0000 0001	1
	1111 1111 1111 1111	2^{16}-1(65535)
int	0000 0000 0000 0000 0000 0000 0000 0001	1
	1111 1111 1111 1111 1111 1111 1111 1111	-1
unsigned int	0000 0000 0000 0000 0000 0000 0000 0001	1
	1111 1111 1111 1111 1111 1111 1111 1111	2^{32}-1 (4294967295)
long	0000 0000 0000 0000 0000 0000 0000 0001	1
	1111 1111 1111 1111 1111 1111 1111 1111	-1
unsigned long	0000 0000 0000 0000 0000 0000 0000 0001	1
	1111 1111 1111 1111 1111 1111 1111 1111	2^{32}-1 (4294967295)

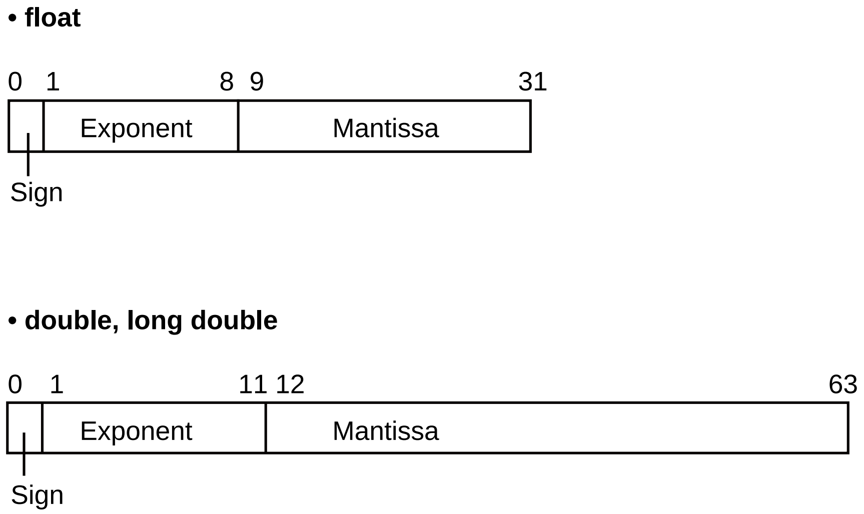
5.3 Floating Types

Table 5.3 shows internal representation of floating type data.

Table 5.3 Internal Representation of Floating Types

Type	Size	Alignment	Signed?	Minimum Value	Maximum Value
float	4bytes	4bytes	Yes	1.17549435e-38F	3.40282347e+38F
double	8bytes	4bytes	Yes	2.2250738585072014e-308	1.7976931348623157e+308
long double	8bytes	4bytes	Yes	2.2250738585072014e-308	1.7976931348623157e+308

A floating-point number is expressed in IEEE standard format. Floating type is expressed in IEEE single-precision format (32 bits); double and long double types in IEEE double-precision format (64 bits). Figure 5.1 shows these internal representations.



Sign : Indicates the sign of floating-point number. When 0, indicates positive. When 1, indicates negative.

Exponent : Indicates the characteristic of the floating-point number as a power of 2.

Mantissa : Indicates the effective value of the floating-point number (binary).

Figure 5.1 Internal Representation of Floating Types

Table 5.4 shows examples of internal representation of floating type data.

Table 5.4 Examples of Internal Representation of Floating Type Data

Type	Internal Representation	Value
float	0011 1111 1000 0000 0000 0000 0000 0000	1.0
	0011 1111 0100 0000 0000 0000 0000 0000	0.75
double, long double		
	0011 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	1.0
	0011 1111 1110 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	0.75

5.4 Arrays

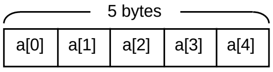
The elements of array type data are continuously allocated to a series of memory locations. The size of array type data is “the size of array element × the number of elements”. The alignment of the array type data is the same as that of the data type of array element. The examples show the following internal presentation of array type data and their alignment :

- **Example 1**

Coding

```
char a[5];
```

Internal Representation



Alignment

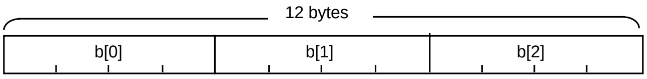
a is 5 bytes (1 byte x 5). Since the data type of the array element is char, the alignment is also char type alignment (1).

- **Example 2**

Coding

```
int *b[3];
```

Internal Representation



Alignment

b is 12 bytes (4 bytes x 3). Since the data type of the array element is pointer, the alignment is also pointer type alignment (4).

- **Example 3**

Coding

```
char c[3][2];
```

Internal Representation

6 bytes

c[0][0]	c[0][1]	c[1][0]	c[1][1]	c[2][0]	c[2][1]

Alignment

`c` is 6 bytes (1 byte x 2 x 3). Since the data type of array element is char, the alignment is also char type alignment (1).

5.5 Structures

The member data of structure type is allocated to continuous locations in the order of member declaration. Particular allocation rule applies to bit-field members (see 5.9 “Bit-Fields”).

The alignment of structure type data is determined by the data type alignment of that member, i.e., the largest alignment. The size of a structure is, in general, the sum of the sizes of that member. Exception: when allocating members of structure, 1-3 bytes of gap may be inserted after the end location of the preceding member due to alignment for each member.

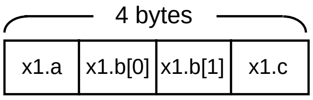
Since the alignment of a structure type is equal to the largest alignment in the member, if the end location of the last member is at an address which does not match the alignment, the location is assumed to end at the address which matches the alignment. Therefore, the size of a structure type is always a multiple of the alignment. Examples of internal presentations of structure type data and their alignment are shown in Examples 1 to 4 :

- Example 1

Coding

```
struct s1{
      char a;
      char b[2];
      char c;
}x1;
```

Internal Representation



Alignment

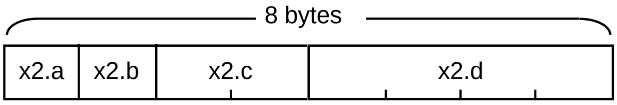
Since alignment of each member is 1 byte, total alignment is also 1 byte. The size of a structure is 4 bytes.

- **Example 2**

Coding

```
struct s2{
      char  a, b;
      short c;
      char *d;
}x2;
```

Internal Representation



Alignment

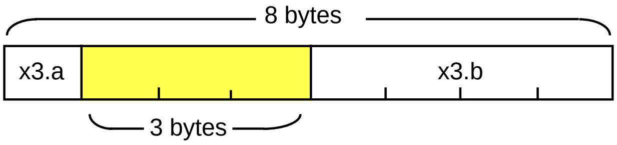
Since a member has 4-byte alignment pointer type, the alignment of the structure is 4 bytes.

- Example 3

Coding

```
struct s3{
      char a;
      int  b;
}x3;
```

Internal Representation



Alignment

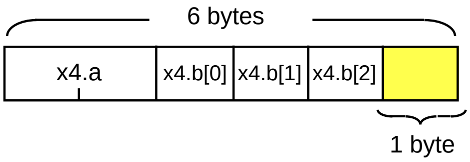
Since a member has 4-byte alignment int type, the alignment of the structure is 4 bytes. Since 3 byte gap is used for alignment of member b, the total size is 8 bytes.

- **Example 4**

Coding

```
struct s4{
      short a;
      char  b[3];
}x4;
```

Internal Representation



Alignment

Since a member has 2-byte alignment short type, the alignment of the structure is 2 bytes. Since the location of the last member ends at odd byte, a 1 byte of memory space is added and the size becomes 6 bytes.

5.6 Unions

The data of members of union are assigned the same address. The alignment of union type data is the maximum value of the alignment of the data type of that member. The alignment of entire union is either 4, 2 or 1 byte(s) depending on the largest alignment among the members.

The size of a union is the maximum size of a member. If the alignment of the union is 2 bytes or 4 bytes and the largest value is not a multiple of alignment, the area of the union is set to a multiple of the alignment. The size of the union is always a multiple of the alignment.

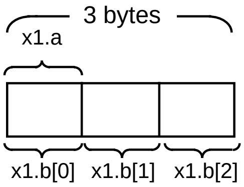
Examples 1 to 3 show the internal presentation of union type data.

- **Example 1**

Coding

```
union u1{
        char a;
        char b[3];
}x1;
```

Internal Representation



Alignment

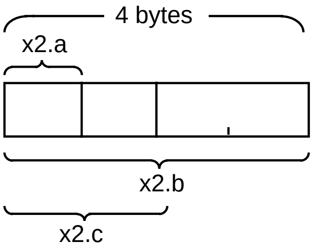
The alignment of each member is 1 byte and therefore the alignment of the entire union is 1 byte. The size is 3 bytes.

- **Example 2**

Coding

```
union u2{
      char  a;
      int   b;
      short c;
}x2;
```

Internal Representation



Alignment

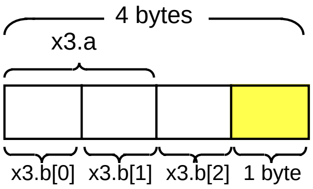
One member has 4-byte alignment and therefore the alignment of whole union is 4 bytes. The size is 4 bytes.

- **Example 3**

Coding

```
union u3{
      short a;
      char  b[3];
}x3;
```

Internal Representation



Alignment

One member has 2-byte alignment and therefore the alignment of whole union is 2 bytes. Since the size of the member having the largest location is odd byte, a 1 byte of memory space is added and the size becomes 4 bytes.

5.7 Enumeration Types

The internal representation of an enumeration type is the same as that for an int type.

Table 5.5 Internal Representation of Enum Type

Type	Size	Alignment	Signed?	Minimum Value	Maximum Value
enum	4bytes	4bytes	Yes	-2^{31}(-2147483648)	2^{31}-1(2147483647)

The value of the enumeration type member's name starts with 0 and subsequent integers in that order with 0 given to the member which first appears.

- Example `enum{ a, b, c }`
  The value of the members: a = 0, b = 1, c = 2

5.8 Pointers

The value of a pointer type data represents the address of the location at which the data or function is stored. The pointer type data is 4 bytes.

Table 5.6 Internal Representation of Pointer Type

Type	Size	Alignment	Signed?	Minimum Value	Maximum Value
Pointer	4bytes	4bytes	No	0	2^{32}-1(4294967295)

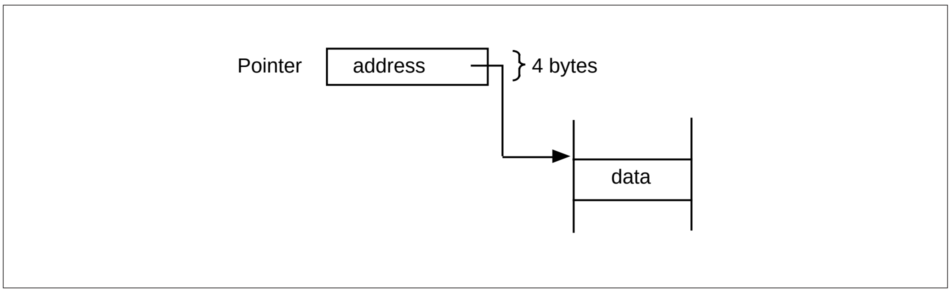


Figure 5.2 Pointer Type Data

||||| Note |||||

The C compiler treats pointer type data as unsigned data. To use pointer type data as a signed logical address value, first convert the data into an int type.

5.9 Bit-Fields

A bit-field is a set of bits. Members in a structure or union have data in the form of bit-field. When declaring a bit-field, specify the field width (number of bits).

The bit-field data are stored into the byte (8 bit), half word (16 bit) or word (32 bit) memory area in the order which they declared.

5.9.1 Data Type for Bit-Field

Valid bit-field data types can be signed or unsigned, char, short, int or long types.

||||| Note |||||

ANSI-C allows only the int (signed int or unsigned int) for bit-field data. This means that a program using char or short type data may not be processed by compilers other than CC32R.

Before being used, a bit-field is expanded to data of the size of the type of which it is declared. The specified bit-field width is allocated with the expanded data bits, low-order bit first. The high-order bit, the expanded field, is set as follows depending on whether the bit-field is signed or unsigned.

- Unsigned bit-field — The high-order bit is set to 0 during promotion (zero extension).
- Signed bit-field — The high-order bit is set to the MSB[Note)] of the bit-field (sign extension).

For example, see "`char member:2`" in Figure 5.3, when using a char type bit-field whose width is declared as 2 bits, high-order 6 bits are sign extended to 8 bits and are processed as a 1 byte data. The value of the byte allocated to the bit-field is "1100 0000". The upper 2 bits, "11", are allocated to the member.

IMPORTANT! : A bit-field, though having the same bit pattern, may take on a different range that it expresses depending on signed or unsigned (see the example below). So be careful.

Example : `char a:4;` —— a ranges between -8 and 7. -1 for 1111.
`unsigned char b:4;` —— b ranges between 0 and 15.
15 for 1111.

Note) MSB = most significant bit

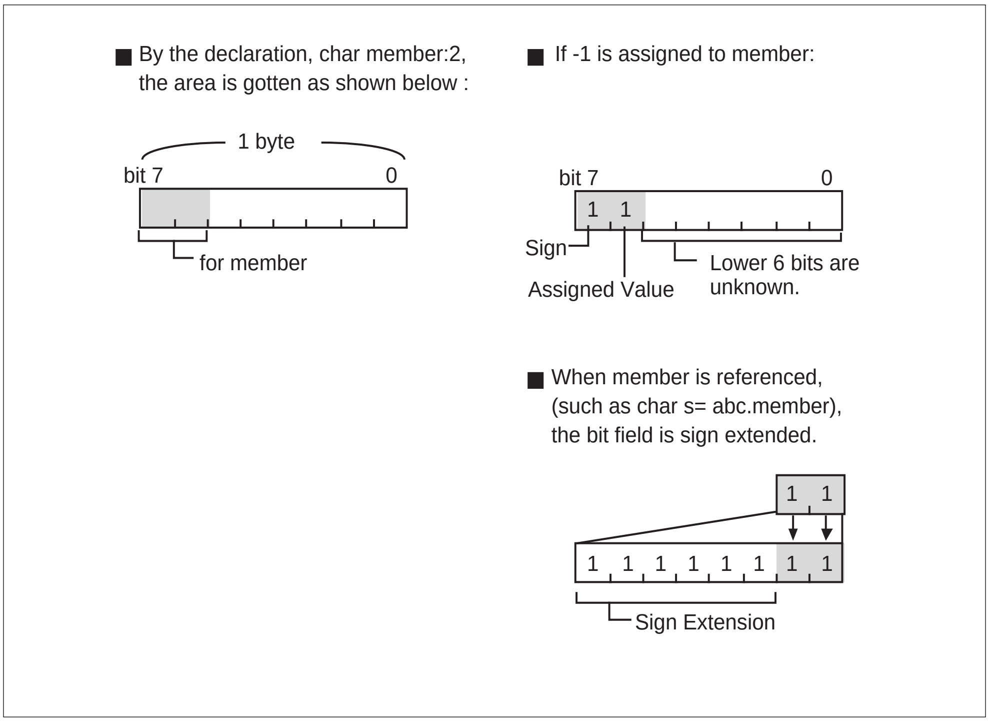

■ By the declaration, char member:2,
the area is gotten as shown below :
1 byte
bit 7
0
for member
■ If -1 is assigned to member:
bit 7
0
1 1
Sign
Assigned Value
Lower 6 bits are
unknown.
■ When member is referenced,
(such as char s= abc.member),
the bit field is sign extended.
1 1
1 1 1 1 1 1 1 1
Sign Extension

Figure 5.3 Example of Internal Representation of Bit-Field

||||| Note |||||

The behavior of a bit-field may depend on an implementation. So be careful of the following in transporting an application or the like :

• int

In ANSI-C, a bit-field declared with “signed int” is sign-extended, whereas whether a bit-field declared with “int” is zero-extended or sign-extended is indeterminate. Thus there can be a chance for a bit-field declared with “int”, though holding a signed data item, to be zero-extended depending on a compiler used.

```
struct b1 {
     signed int          a:4;
     int                 b:4;
}x;
```

`x.a=-1;` /* The value stored in `x.a` becomes -1. */

`x.b=-1;` /* The value stored in `x.b` is indeterminate. It correctly becomes -1 by sign-extension, but becomes 15 by zero-extension. */

• signed int

In ANSI-C, a bit-field declared with "signed int" is sign-extended. Thus specifying 1 for the field width allows the value to range between -1 and 0, not between 0 and 1.

`singed int c:1;` /* `c` can take on a range between -1 and 0. */

5.9.2 Packing and Alignment

The size of the type specifiers that can be used for bit-fields is signed or unsigned 1 byte (8 bits, char), 2 bytes (16 bits, short) and 4 bytes (32 bits, int or long).

Allocation of a bit-field to a memory location is shown in examples below :

- Size of the alignment boundary at location.

  Memory space to accommodate all members are reserved in units of size (1, 2, and 4 bytes) denoted by the type specifiers. For example, memory space is divided in 4 byte locations in the case where the structure is composed of int type bit-fields.

- Bit-field is packed into the memory space denoted by the type specifier.

  A series of bit-fields are packed into a unit memory space (1, 2 or 4 bytes) as long as they can go. (See Example 1.) For example, an int type structure consisting of 3 members of field width 4 (12 bits) is completely packed into a 4-byte location (32 bits).

- Arrangement is from upper to lower bits.

  A series of fields are placed, in the order of declarations, starting with the left side (upper bit) to the right (lower bit) of the memory location whose size is specified by the type specifier.

- No field can exist across an alignment boundary

  No bit-field is separated by an alignment boundary. If not all bits of a field is accommodated in a location, the field is shifted to the next location, leaving unused bits in the preceding location. This means that the data string is discontinued. (See Example 2.)

- 0-bit-field is placed on the next location

  If the size of the next field is 0 bit, the field is placed at the next location. (See Example 3.)

- Different alignments for different type bit-fields

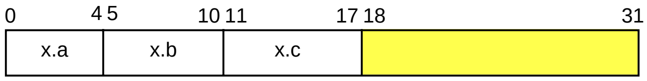
  If a structure or union is composed of bit-fields (members) of different types, every time a field is placed, the size of the type specifier of that bit-field is recognized as the alignment of the memory location. The size of the alignment of the structure and union is equal to the size of the type which has the largest size. (See Example 4.)

- **Example 1**

Coding

```
struct b1 {
          unsigned int a:5;
          unsigned int b:6;
          unsigned int c:7;
} x;
```

Internal Representation



Description

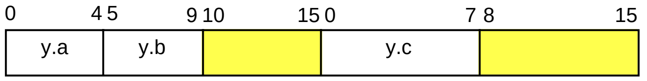
The member of structure b1 is stored at the leftmost and subsequent 4 byte location (size of unsigned int).

- **Example 2**

Coding

```
struct b2 {
          short a:5;
          short b:5;
          short c:8;
} y;
```

Internal Representation



Description

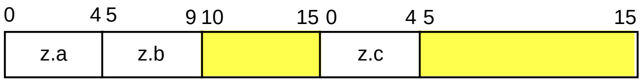
The c of the structure b2 would overflow from the third location (6 bits) and is allocated a new location.

- **Example 3**

Coding

```
struct b3 {
          short a:5;
          short b:5;
          short:0;
          unsigned short c:5;
} z;
```

Internal Representation



Description

The 0-bit bit-field (short: 0) of structure b3 is placed before `c` which is then allocated a new location.

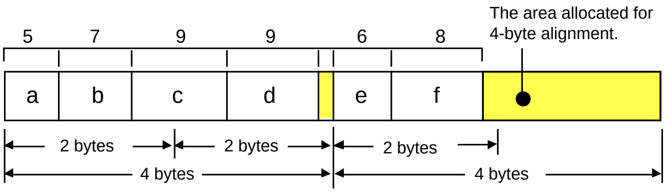
- **Example 4**

Coding

```
struct b4 {
          short a:5;
          short b:7;
          int c:9;
          int d:9;
          short e:6;
          short f:8;
 } z;
```

Representation

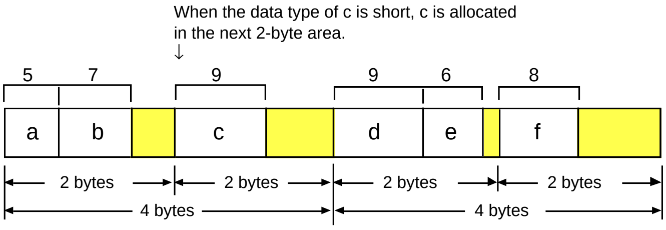
Internal



Description

In the structure b4, different bit-field type specifiers are used. The largest type specifier is int and the entire alignment is 4 byte (32 bit) long. The type specifier for each field specifies the bytes of the location at which the bit-field is to be placed. Because `c` is int type, the alignment is 4 bytes. The 9 bits can follow `b`.

If c is short type, the alignment is 2 bytes. There are 4 bits left following b in that 2-byte location, the 9-bit long c is not accommodated. c is stored in the next 2-byte location. If c is short type, it is stored in the next 2-byte location (see figure below).



Chapter 6

C Calling Conventions

This chapter describes the way C compiler calls a C program, and how to interface with an assembly program.

6.1 Register Usage

6.1.1 General Register (R0-R15) Usage

The M32R is provided with 16 32-bit general registers (R0-R15) (see Figure 6.1). The C compiler uses R14 as the link register; R15 as the stack pointer (SP); and R0-R13 as working register to temporarily store intermediate results of function operation and the variables.

Registers R11-R13 can also be used as the base register when declared to do so in the program (in this case R11 to R13, as specified in the base register, always store the base addressR11 to R13, as specified in the base register, always store the base address.).

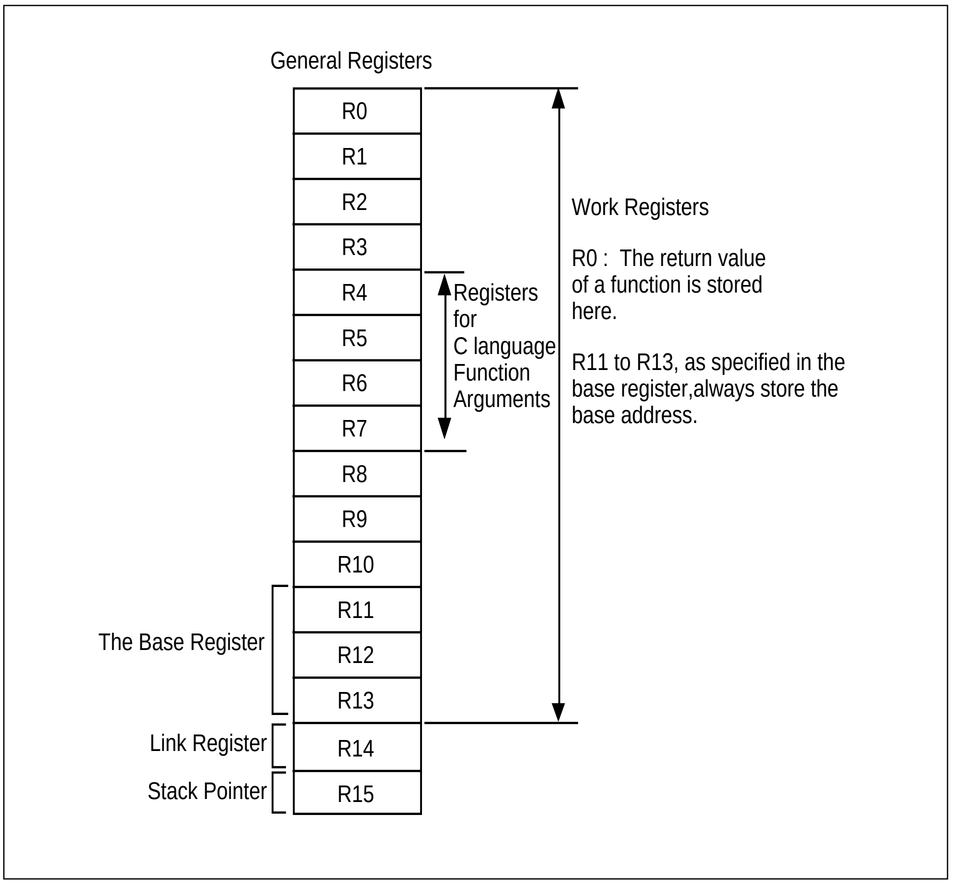

General Registers
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
Registers for C language Function Arguments
Work Registers
R0 : The return value of a function is stored here.
R11 to R13, as specified in the base register,always store the base address.
The Base Register
Link Register
Stack Pointer

Figure 6.1 General Registers Used

The applications and associated operations of the general registers used by the C compiler are described below.

- Work registers (R0-R13)

  Store intermediate results of function operation and variables. The R0 stores the values returned by function.

- Registers for function arguement (R4-R7)

  The registers R4-R7 are used to store function argument(C language).

- Registers for the Base Registerf (R11-R13)

  The registers R11-R13 are used to the Base Register(See Appendix A).

- Link register (R14)

  The R14 stores the return address during function call. It may also be used as work register or register variable register by the C compiler.

- Stack pointer (R15)

  The R15 is a stack pointer (SP) which stores the lowest address of the stack area. The SP manages the stack.

6.1.2 Register Consideration

Table 6.1 shows whether the contents of register is retained or not when a function is called by the C program. "Yes" means that the register is saved and restored automatically by a called function, and "No" shows that you must save and store the register when you call a function.

Table 6.1 Is State of the Register Assured?

Register(s)	Consideration	
R0~R3	No	
R4~R7	No	For Function Argument
R8~R10	Yes	
R11~R13	Yes	For the Base Register
R14 (Link register)	No	
R15 (SP)	Yes	
PSW(CR0)[Note1]	Yes	
CBR(CR1)[Note2]	Yes	
Accumulator	Yes	

Note 1) PSW(CR0): Processor post status register. An M32R control register in which conditional bit of stack mode, interrupt enable and operation result, and saved value of them are set.

Note 2) CBR(CR1): Conditional bit register. An M32R read only control register which stores the results of the preceding operation (carry, borrow, overflow, etc.).

6.2 Stack Frame Configuration

The stack frame is an area allocated on the stack each time a function is called. The Figure 6.2 shows a typical stack frame (areas (0) to (2) in the order from higher to lower address). The argument area of area (0) is pushed to the stack only under the conditions described in 6.4.2, "The Cases where Stack-passing is Valid."

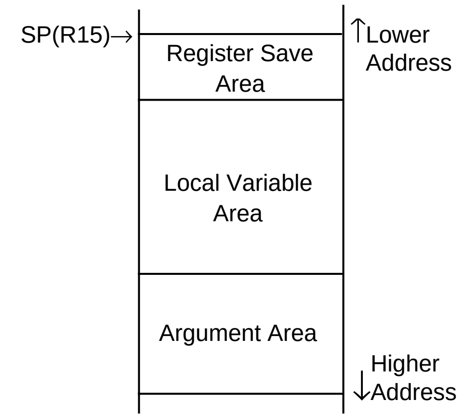
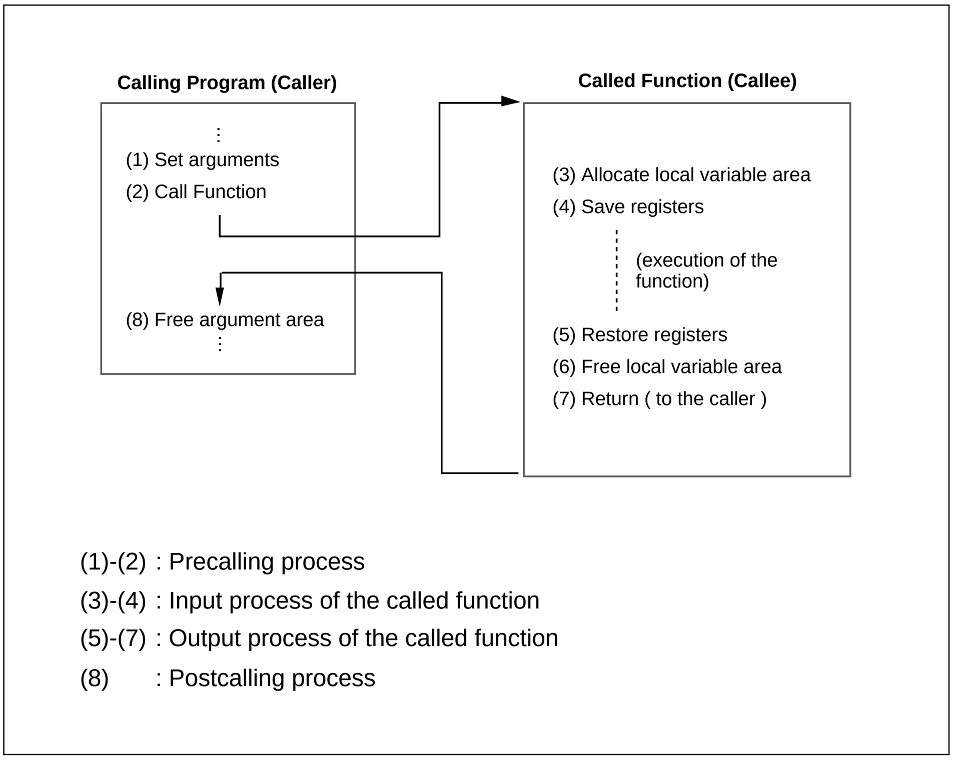


Figure 6.2 Typical Stack Frame

(1) Argument area	Case of the Stack-passing, arguments to be passed to the function called are set in this area. Arguments are arranged so that the first argument is at the lowest address.
(2) Local variable area	The area into which the local variable declared by the called function is to set.
(3) Register save area	Used to save the current contents of the general register which will be used by the function called.

6.3 Call and Return Procedures

Calling and returning of a function by C program are performed in the order given in Figure 6.3.



(1)-(2) : Precalling process
(3)-(4) : Input process of the called function
(5)-(7) : Output process of the called function
(8) : Postcalling process

Figure 6.3 Function Call and Return in C

(1) Setting arguments

C compilers generally use "register passing" to pass arguments when a function is called. Note that type conversion and alignment rules apply to how the content of arguments is stored (see 6.4, "parameter register").

(1-1) "Register passing"

The first four arguments are stored sequentially in registers R4 to R7. If the function takes fewer than four arguments, the number of registers equivalent to the number of arguments is used. (For example, only R4 and R5 are used in the case of two arguments.) Also, if there are five or more arguments, stack passing is applied to the fifth and subsequent arguments.

(1-2) "Stack passing"

The following applies if the conditions in 6.4.2, " The Cases where Stack-passing is Valid" are satisfied.)

The arguments are pushed onto the stack and the argument area is set up. The arguments are stacked in sequence from the last argument. The first argument is set at the lowest address in the argument area, as shown in Figure 6.4.

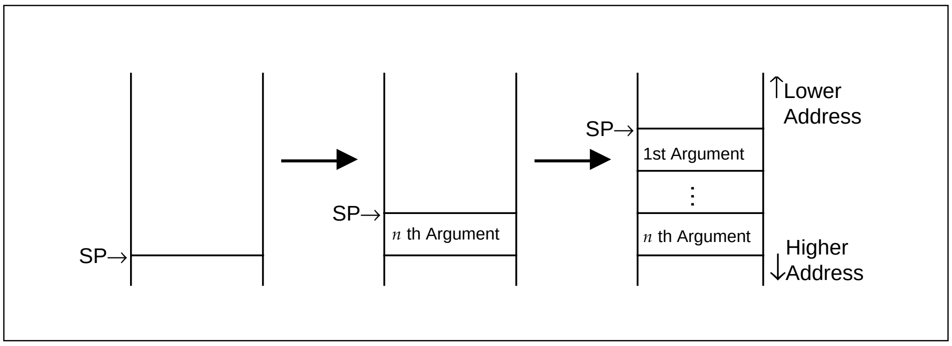


Figure 6.4 Argument Setting

When a called function returns the structure or union type return value, after the first argument is pushed, the top address of the area where the return value is set is pushed . The return value setting area (generally, in the stack area) is reserved by the calling side before starting the function call. Figure 6.5 shows the argument area for the function having structure or union type return value.

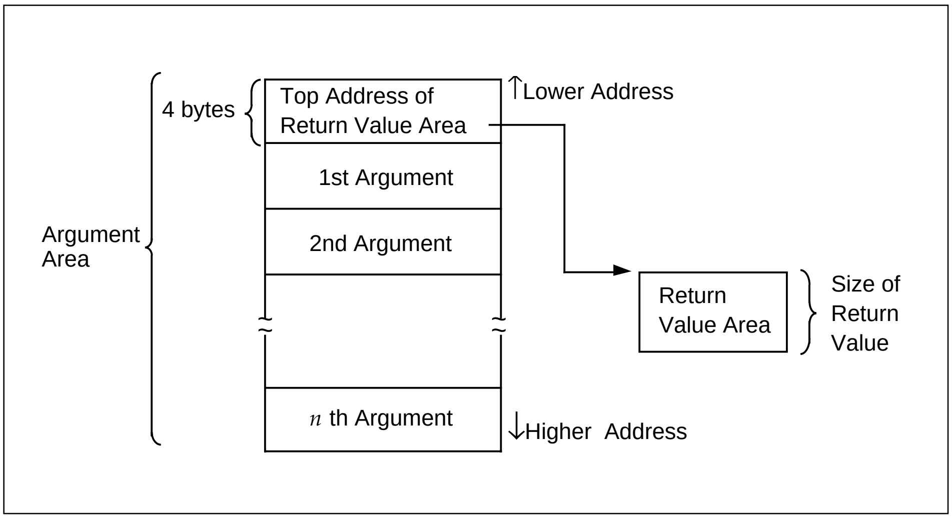


Figure 6.5 Argument Area for the Function which Returns Structure or Union

(2) Call function

This function is called by BL or JL instruction. The address (return address) of the instruction following the function call instruction is set in the link register R14.

(3) Allocate local variable area

On the stack, a local variable area is allocated, which the called function will use. The size of the reserved area is subtracted from the SP value. The size of the reserved area is always a multiple of 4.

(4) Save registers

The contents of registers (R8-R13) to be used by the called function are saved on the stack. The size of the saving area is “the number of registers to be saved ¥ 4 bytes”. When the link register R14 is used, then the link register is saved.

(5) Restore registers

The register(s) saved at the entry of a called function (see “(4)Save registers ”above) is restored.

(6) Free local variable area

The local variable area reserved at the entry of a called function (see “(3) Allocate local variable area” above) is released. That is, the size of the area is added to the SP**.**

(7) Return (to the caller)

The return address stored in the link register is reset in the Program Counter, returning process control to the calling side program.

(8) Free argument area

Once the control is returned. the size of the argument area is returned back to the SP, releasing the argument area.

6.4 Parameter Passing

C compilers generally use "register passing" to pass arguments when a function is called. (If the function takes five or more arguments, or if the function's arguments include floating point types (double, long double), structure types, or union types, etc., stack passing is used. The following describes how to set the arguments.

6.4.1 Rules Parameter Passing

By register-passing, actual arguments to a function are passed via the registers R4 to R7. The first 4 arguments are stored in R4-R7 one by one in the order declared by the functions. For example, if only two arguments are declared, only two registers, R4 and R5, are used. Once the function is called by register-passing, the values in R0-R7 are not guaranteed.

The arguments that can pass into the registers are those of character, integer, pointer and float types.

Example : `int func(a, b, c);`

Executing this function call stores the arguments;

R4	a
R5	b
R6	c

- Type conversions

  If the type of an argument has been declared by the function prototype declaration, the argument is converted to that type. Otherwise, the rules given below are applied :

  - char, unsigned char, short, or unsigned short is promoted to int.
  - float is promoted to double [*].
  - Otherwise unconverted.

- Store into the register

  Arguments which have done type conversion as above are stored into the registers follows depending on their types :

  - If the type after conversion is one of char, unsigned char, short and unsigned short type, the argument first is converted to type int and then is pushed onto the stack.

[*] float type arguments are converted to double types and pushed to the stack if their type has not been declared in a prototype declaration.
To use register passing with float type arguments, be sure to declare their type in a prototype declaration.

6.4.2 The Cases where Stack-passing is Valid

Part of or all arguments are passed through the stack in the following cases :

- If there are five or more arguments, stack passing is applied to the fifth and subsequent arguments.
- If the first 4 arguments include either floating type, structure type or union type, the arguments up to the argument preceding the argument of such a type are passed by registers. The remaining arguments are passed by the stack.
- If the return value is either floating types, structure type or union type, all the arguments are passed by the stack.
- If the function has a variable parameter ("..." is specified at the end of the parameter list, as with the printf function), the variable argument and the preceding argument are passed by the stack.

6.4.2.1 Pushing onto the stack

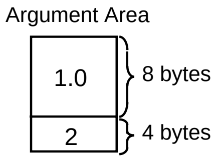
Arguments which have done type conversion as above are pushed onto the stacks :

- If the type after conversion is one of char, unsigned char, short and unsigned short type, the argument first is converted to type int and then is pushed onto the stack. If an argument has another scalar type, it pushed on without being converted. An argument which has converted to type float will not be converted to type double when pushed on the stack.
- If an argument has type struct or union with 4-byte boundary alignment is directly stacked. Otherwise (i.e., it is not a 4-byte boundary), first the area with 4-byte boundary which can be represented the argument is allocated on the stack, and then, the argument is set to the lowest address (a multiple of 4) and subsequent of that area.

The examples, 1 to 5 shown below, describe various stack-passing procedures (figures are images of the argument are on the stack) :

- **Example 1**

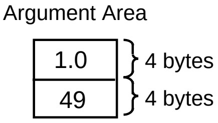
```
int f();
       ⋮
       f(1.0f, 2);
```



The first argument is type float without the function prototype declaration. It is therefore first converted to type double and then is pushed on the argument area on the stack.

- **Example 2**

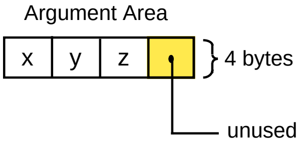
```
int f(float, char);
            :
       f(1, 49);
```



The first argument is converted to 1.0 as type float. The second argument is first converted to type char according to the function prototype declaration and then is reconverted to type int as it being pushed on the stack.

- **Example 3**

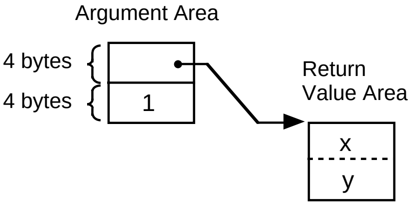
```
struct s{
        char x,y,z;
  }a;

  int f();
        :
     f(a);
```



The argument s requires 3 bytes size and is aligned with 1 byte boundary. When the argument is pushed on the stack, a single byte of unused area is added to the argument area on the stack to make its size 4 bytes.

- **Example 4**

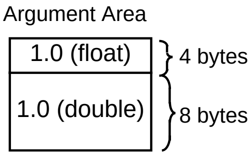
```
struct s{
   int x, y;
}f();
     :
 f(1);
```



When the function returns struct, the pointer (address) to the return value area (reserved by the calling side) is set to the low address of the argument area on the stack.

- **Example 5**

```
int f(float, ...);
             :
    f(1.0, 1.0f);
```



Since the first argument is denoted as type float by the function prototype declaration, the argument is converted to type float. The second argument is not declared its type and is therefore converted to type double.

6.4.3 Function Names after Compiling

Function names in the object module are named by preceding the names of functions in the C program with a dollar mark ($). However, in the case of functions that return floating-point types (double or long double), structure types, or union types, and in the case of functions that have variable arguments, the following rules apply:

- Functions which return either floating-point number, structure or union :
  The function name is preceded by an underscore (_).
- Functions having variable parameters :
  Assume that the number of arguments passed via register is *n*,

When *n*=0 : The name is a function name used by the C program with a preceding underscore (_).
Example : `int foo(char *, ...);`
is given the name `_foo`.

When *n*=4 : The name is a function name used by the C program with a preceding dollar mark ($).
Example : `int foo(int, int, int, int, int,…);`
is given the name `$foo`.

When *n*=1, 2 or 3 :
The name is a function name used by the C program with a preceding dollar mark ($) and the number of arguments passed by register-passing.
Example : `int foo(int, int,…);`
is given the name `$1foo`.

6.4.4 How to Refer set Arguments

To refer to an arguments within the called function, refer to the stack frame if the argument is passed through the stack, or refer to the register into which the argument is set when the argument is passed through the register. Note that some arguments are passed through stack even if register- passing is specified (e.g., structure type) . In such a case, refer to the stack frame.

6.5 Setting Return Value

The return value (from the function) is first converted into a value of the type suitable for return.

- Integer, pointer — The called side set the return value to R0. The calling side refers to R0.
- Floating-point number — Same as for the structure and union types.
- Structure, union — The calling side set the address of the return value setting area as the first argument (see Figure 6.5 and Figure 6.6). The called side set the return value by using this address. The calling side refers to the return value setting area.

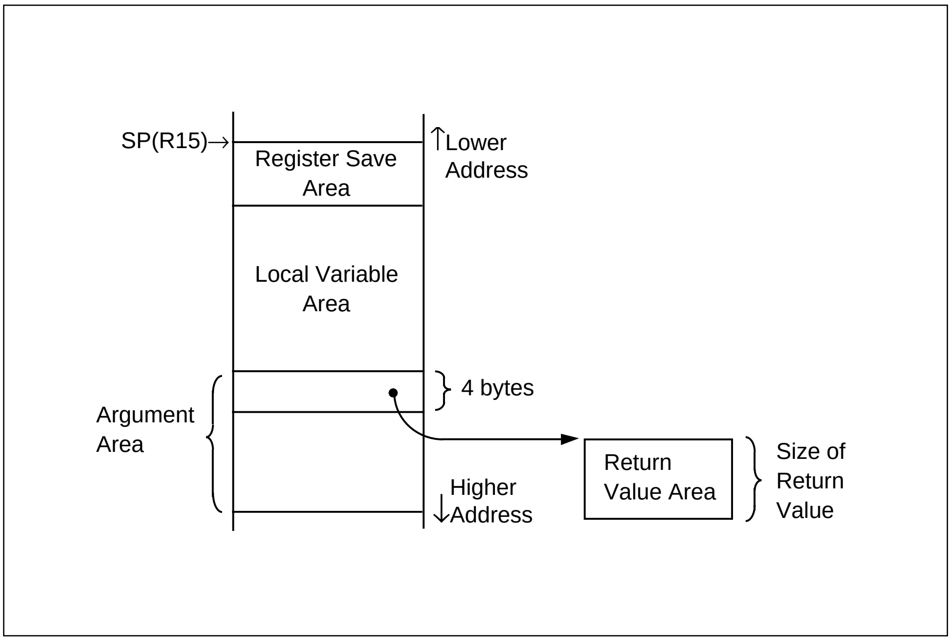


Figure 6.6 Struct or Union Type Return Value Setting

6.6 Interface with Assembly Program

When developing an application by linking more than two programs (modules), a program may refer to data of a different program, or may call a function from another program. This section describes how a C program can refer to an assembly program and how the assembly program can refer to the C program.

To call C program function from a different C program, see 6.3 “Call and Return Procedure”. For data reference, see 6.4 “Parameter Passing”.

To reference C program data from an assembly program, or to call a C program function from an assembly program, it is necessary to write a program by following the C calling rule.

6.6.1 Referencing Assembly Data from a C Program

To reference assembly data from a C program, write programs by following the example described in Figure 6.7.

C program

```
extern int i, j;

func()
{
   i = j;
}
```

Assembly program

```
     .EXPORT   _i, _j
     .SECTION D,DATA,ALIGN=4
_i:  .DATA.W   1
_j:  .DATA.W   1
     .END
```

Figure 6.7 Referencing Assembly Program Data

- In an assembly program

  Declare the C program data to be referenced in a pseudo-instruction, .EXPORT or .GLOBAL to enable external reference. In the example in Figure 6.7, declare labels _i and _j in .EXPORT.

- In an C program

  Specify the assembly program data to be referenced by using an external declaration. When declaring, use a corresponding assembly program label name but remove the underscore (_) from the name.

6.6.2 Referencing C Data from an Assembly Program

To reference C data from an assembly program, write an assembly program by following the example in Figure 6.8.

C program

```
    :
char a ,b;
    :
```

Assembly program

```
.GLOBAL   _a
.GLOBAL   _b
.SECTION P,CODE,ALIGN=4
LD24 R1,#_a
LDB  R1,@R1
LD24 R0,#_b
STB  R1,@R0
JMP  R14
.END
```

Figure 6.8 Referencing C Program Data

- In the assembly program

  To reference char type variables a and b written in C from the assembly program, specify labels `_a` and `_b` by the pseudo-instruction, .IMPORT or .GLOBAL.

The CC32R prefixes an underscore (_) to the external reference and the definition symbol of C upon outputting them to the object module. The variables in the C program in Figure 6.8, for example, correspond to the labels `_a` and `_b` in the assembly program.

6.6.3 Calling Assembly Routines from a C Program

To call an assembly function from a C program, write programs by following examples described in Figure 6.9.

Calling program (C program)

```
extern int func();

main()
{
     func(1,2);
}
```

Called program (assembly program)

```
.EXPORT   $func
         .SECTION P,CODE,ALIGN=4
$func:
         MV  R0,R5   ;Transfer content of the 2nd argument
                     ;to R0
         ADD R0,R4   ;Set result of the 2nd argument
                     ; + the 1st argument to R0
         JMP R14
         .END
```

Figure 6.9 Calling Assembly Program Function

- As passing arguments

  A symbol name of the assembly program is the corresponding function name but preceded by the dollar mark ($).

  - Calling program (C program)
    Arguments are set to the registers.

  - Called program (assembly program)
    In the example shown in Figure 6.9, the program references the first argument by R4 and the second argument by R5. It sets the return value to R0.

6.6.4 Calling C Routines from an Assembly Program

To call a C program function from an assembly program, write an assembly program by following the example described in Figure 6.10.

Calling program (assembly program) - register-passing

```
.GLOBAL   $func
          .SECTION   P,CODE,ALIGN=4
          LDI R5,#2            ;y
          LDI R4,#1            ;x
          BL  $func
          .END
```

Called program (C program)

```
func(int x, int y)
{

}
```

Figure 6.10 Calling C Program Function

- As passing arguments

  A symbol name of the assembly program is the corresponding function name but preceded by the dollar mark ($).

  The calling program (assembly program) sets arguments to the registers (R4-R7). In the example shown in Figure 6.10, arguments are two, so that the first argument is set to R4 and the second to R5.

Chapter 7

Embedded Applications Programming

This chapter provides information necessary to program embedded application programs (embedded applications) for the M32R system. For further information on the link function, refer to "CC32R User's Manual <Assembler>", Chapter , "Linker lnk32R".

7.1 Compiler-Generated Sections

When an object module file is generated by the C compiler using a C source file, all codes and data contained in the program are automatically defined as one of the following sections :

• The P section	Program code area.
• The C section	Constant data area (for `const`-declared variables).
• The D section	Data area with initializers (for global variables having initial values).
• The B section	Data area without initializers (for uninitialized global variables).

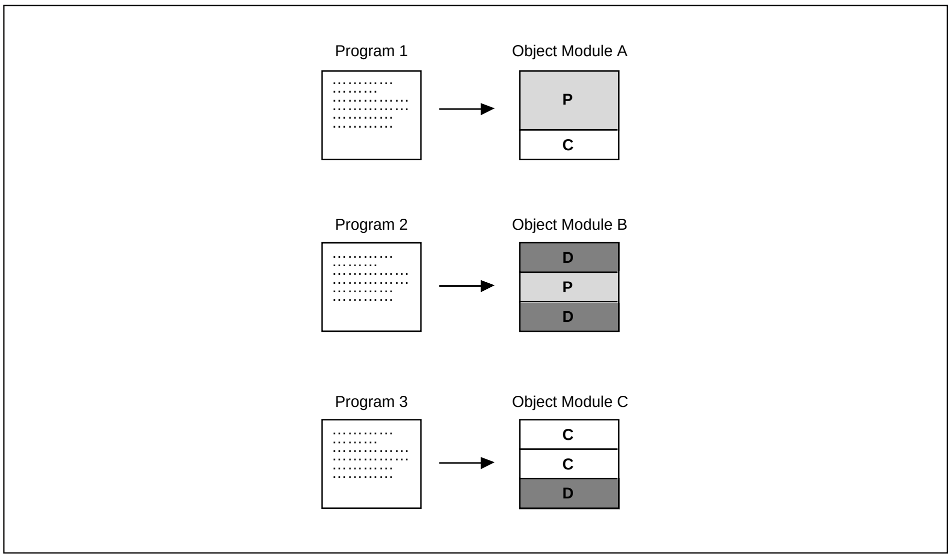


Figure 7.1 C Compiler Defines a Program to Sections

A section is a unit program processed by linker. Each section has the section name, section attribute and location attribute, as shown in Table 7.1 [Note1].

Table 7.1 Sections Output from C Compiler

Section Name	Section Attribute	Location Attribute	Description
P	CODE	ALIGN=4	Program code area
C	DATA	ALIGN=4	Constant data area (for `const`-declared variables).
D	DATA	ALIGN=4	Data area with initializers (for global variables having initial values excluding `const`-declared variables).
B	DATA	ALIGN=4	Data area without initializers (for uninitialized global variables).

The linker assumes the sections are the same section when these sections have the same name and attributes. It links these same sections[Note2] and arranges them continuously in the load module as shown in Figure 7.2.

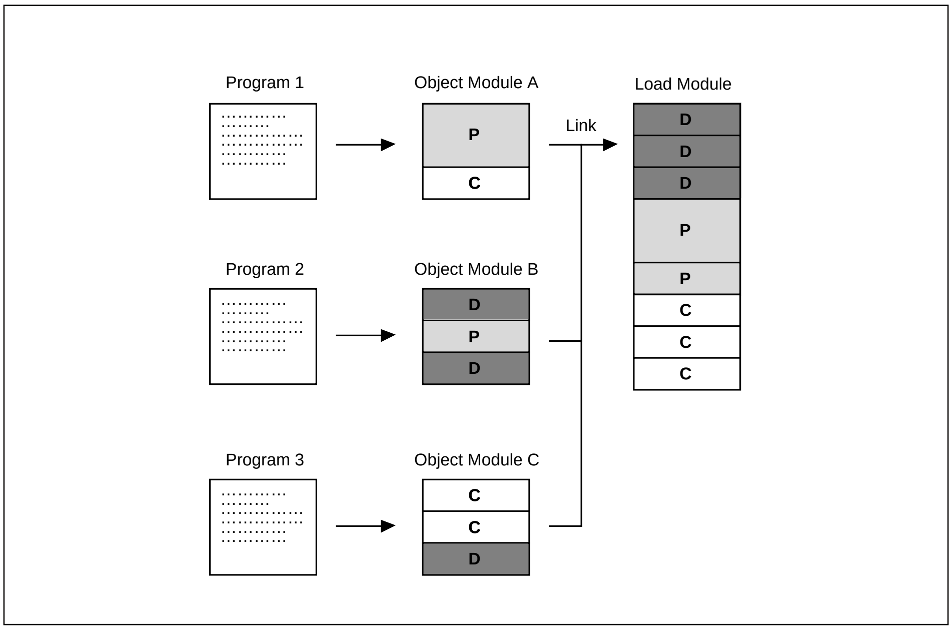


Figure 7.2 Linking Sections by Sections (Image when "`-SEC D,P,C`" is specified)

[Note1] The section name and attributes can be defined as other than those in Table 7.1 by using the pseudo-instruction .SECTION (assembler- supported). The use of .SECTION are described in "CC32R User's Manual <Assembler>". For details on section name and attributes, see "CC32R User's Manual <Assembler>", in Chapter lnk32R.

[Note2] Linking method varies with section attribute and linking order varies with the command options.

With the C compiler, section linking order and the start address of each section can be specified by using the command option -SEC or -MEM (equivalent to -LOC used for direct startup from linker). The linker determines the address of each section by following these specifications.

These features are useful when developing embedded application or writing the application program into ROM. For further information, refer to the "CC32R User's Manual 3 - The Others", Chapter Linker.

||||| Note |||||

Options -SEC and -MEM cannot be specified simultaneously.

In practice, the linking order and start address of each section are determined by the conditions described in this section. For function of options and use of options, refer to Chapter 3 "Invoking the Compiler".

- **Section linking order**

  When the linking order is specified, the sections are linked in the order of priority levels. Sections having no linking priority are linked following the lowest level section.

  (1) When specified by the -SEC or -MEM :

  When specified by the -SEC option

  Sections are linked in the order -SEC option is written.

  When specified by the -MEM option (-LOC in the case of linker)

• RAM area	D→B
• ROM area	P→C→D

  (2) Without order setting (default) :

  Sections are linked in the order of the files specified (written) in the command line: If a specified file contains two or more sections, the are linked in the order they appear (written).

- **Start address**

  Specified addresses have priority. Unspecified sections are assigned relative address according to the linking format. Default alignment is 4 bytes (any value can be specified by using the assembler pseudo-directive .SECTION). The start address of a load module is 0 if not specified.

  (1) When specified by the -SEC or -MEM :

  When specified by the -SEC option

  The address is specified for each section

  The start address of section having unspecified address is automatically specified according to the linking order.

When specified by the -MEM option (-LOC in the case of linker)

• D section	The address specified for RAM area
• P section	The address specified for ROM area
• B, C, D section	The address immediately following the end address (lower address) of the preceding linked section.

(2) When not specified (default) :

• Each section	The address immediately following the end address (lower address) of the preceding linked section.
• Entire load module	Address 0

7.2 Embedded Application Programming Procedure

When programming an embedded application, the start-up program and lower level libraries must be developed as well as the user program which performs the desired functions. To link the application, arrange specification and entry point specification suitable for embedding and allocation of address for burning into ROM are required. The load module file must be converted into an S-format file for burning it into ROM. These activities are included in a development of an embedded application and must follow steps (1) to (6) as shown in Figure 7.3.

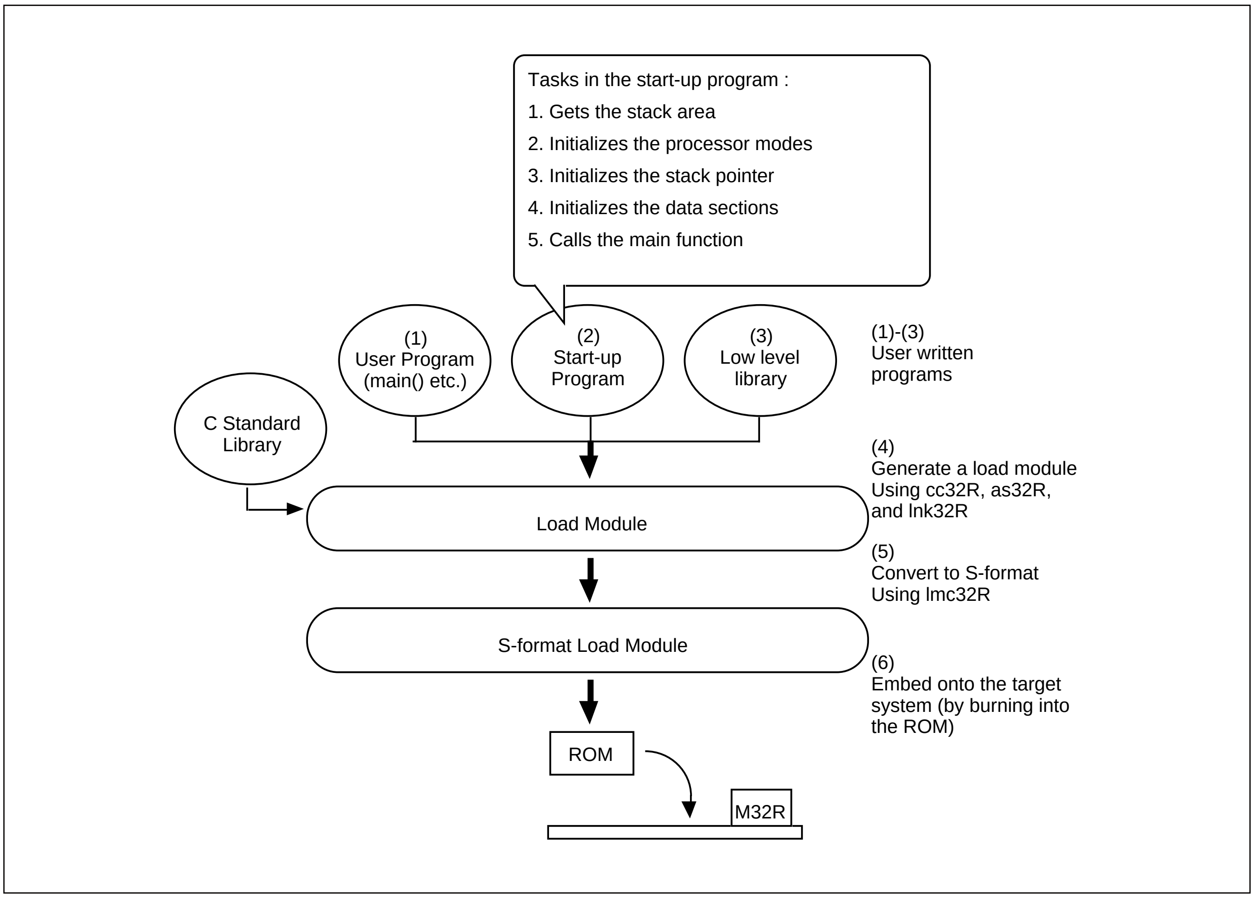

Tasks in the start-up program :
1. Gets the stack area
2. Initializes the processor modes
3. Initializes the stack pointer
4. Initializes the data sections
5. Calls the main function
(1) User Program (main() etc.)
(2) Start-up Program
(3) Low level library
(1)-(3) User written programs
C Standard Library
(4) Generate a load module Using cc32R, as32R, and lnk32R
Load Module
(5) Convert to S-format Using lmc32R
S-format Load Module
(6) Embed onto the target system (by burning into the ROM)
ROM
M32R

Figure 7.3 Embedded Application Programming

(1) Write user program

Write the main function and other routines which perform the required processes. To write in C, follow the C specification (refer to Chapter 4 “C Language Elements”). To write in assembly, follow the M32R assembly specification (refer to “CC32R User’s Manual - 2 Assembler”). Some processes can be performed by writing assembly instructions in the C source program (in-line assembling function).

(2) Write the start-up program

Write a start-up program for the target system. The start-up program performs initialization to allow the application to run on the target system and generally performs the following processes :

1. Gets the stack area
2. Initializes the processor mode
3. Initializes the stack pointer
4. Initializes data area sections
5. Initializes the C standard library
6. Calls the main function
7. Terminates the C standard library

Perform processes 1 to 4 above from the assembly program and 5 to 7 from assembly or C program. For writing the start-up program, refer to 7.3 "Tasks in the Start-up Program".

(3) Write the low level library

If the application performs the following tasks by using C standard libraries, low level libraries should be prepared.

- Standard input and outputs
- Memory management
- Signal handling
- Time management

For information on the low level function(s) which is required by each C standard library function, and on the specifications of the low level functions, refer to Chapter 11 "Low-Level Library". Programming by following these specifications, use the librarian to make these programs libraries.

(4) Generate a load module

Using the cc32R command, prepare load module based on generated user program and start-up program.

When linking, files are referenced in the order specified by the command line and the sections are linked in that order. When writing the input files into the command line, be sure to specify the start-up program in the beginning.

Also specify options for linking as follows :

- -l and -L
  Specify linking of low level library and C standard library.

- -e
  Specifies the start-up address of the start-up program as the entry point.

- -SEC (or -MEM)
  Specifies the order and address of the section allocation. With an embedded application, in general, allocate the D and B sections onto the RAM area, D section initialization data and the P and C sections onto the ROM area. To allocate the data used to initialize the D section onto the ROM area, use linker's "Initial value data sampling function" (the section is named ROM_D).

If the execution of a cc32R command results in a compile error, correct the source file. When all error causes are removed, the linker automatically starts and the load module is generated according to linking specified by the command option(s).

The following example shows the commands specified in the cc32R command line (% is a prompt) :

Example :

```
% cc32R -l m32RcR.lib -e startup
-SEC @D=1000,B,SPI,SPU,P=8000,C,D
start.ms initlib.c user1.c user2.c
```

This example assumes the following input files :

- `start.ms`
  A start-up program initial setting process corresponding to the target system (see samples in 7.3.9).

- `initlib.c`
  A start-up program which calls the main function (see samples in 7.3.9).

- `user1.c user2.c`
  User program including the main function.

When executing the command line of this example, the load module a.mout reflecting the following specifications is generated :

- `-l m32Rc.lib`
  C standard library m32Rc.lib is linked.

- `-e startup`
  Entry point is the start address of the start-up program (when the start address is defined by the symbol start-up in the start-up program). The application is then executed from the start-up program.

- `-SEC @D=1000,B,SPI,SPU,P=8000,C,D`
  The linking order of sections is D→B→SPI→SPU→P→C→ROM_D (SPI and SPU sections are sections prepared for stack by the user in the assembly program. ROM_D section is the data area to initialize D section). The sections are arranged as follows :

D	Only area is located starting at address 1000_{16}
P	Located starting at address 8000_{16}
B, SPI, SPU, C, ROM_D	Located following the preceding section

- `start.ms, initlib.c, user1.c, user2.c`
  After each of these files is compiled and/or assembled, they are linked.

(5) Convert to S-format file

Convert the load module into S-format to write it into the ROM. Use the load module converter lmc32R.

Example : `% lmc32R -d 4 -o file a.mout`

The load module a.mout is converted into S-format and divided into four files, file.m40, file.m41, file.m42 and file.m43. For operation of the load module converter, refer to “CC32R User’s Manual 3 - The Others”, Chapter Load Module Converter.

(6) Embed onto the target system

Using a ROM programmer, write the S-format load module program into the ROM. Install the ROM on the target system.

7.3 Programming the Start-up Program

7.3.1 Tasks in the Start-up Program

To run an embedded application on a target system, "start-up program" is required that is executed before and after a user program (starting with the main function) is called. The start-up program carries out initialization and termination — the steps (1) through (5) given below.

(1) Gets the stack area
(2) Initializes the processor modes
(3) Initializes the stack pointer
(4) Initializes the data sections
(5) Calls the main function

These tasks are basic . The user program and your environment may require adding other tasks and / or removing some of them.

Here follow details of respective steps, and examples of the start-up program are given in Section 7.3.7.

7.3.2 Getting the Stack Area

Get the stack area necessary for the application to run. First declare the stack section area in the assembly program by using pseudo directive .SECTION, and then reserve the stack area by using pseudo directive .RES.

Every time the C program calls a function, stack frame is created on the stack (see 6.2). The stack frame is saved until the called function returns process to the calling program. Therefore, if functions are called successively without returning to their caller such as "the function A → function B → function C" , all stack frames for these functions will be saved when the functions C is called.

Determine the stack size necessary for the application to run by taking into considerations these function calling behavior and stack size necessary for these functions. Since the size cannot be exactly determine, estimate approximate size by referring to Chapter 5 "Internal Data Representation" and Chapter 6 "C Calling Conventions". Most practical way is to actually monitor the stack while running the program for debugging and evaluating purposes (reserving enough stack area at initial stage or measuring stack area space with the debugger etc.). Provide the stack area needed for interrupts in the similar way.

7.3.3 Initializing the Processor Modes

Specify the stacks and interrupt levels for the target microprocessor (register PSW setting). For information on settings (PSW configuration), refer to the microprocessor manual.

7.3.4 Initializing the Stack Pointer

Set the highest address of the reserved stack area as the stack pointer.

7.3.5 Initializing the Data Sections

Embedded application requires the initial setting of the data area (sections D and B). Perform the following steps during application linking and running of the start-up program.

- Tasks during linking of the embedded application
  - Allocate area of sections D and B on the RAM area (no data output).
  - Allocate initial value data of section D (ROM_D) on the ROM area.

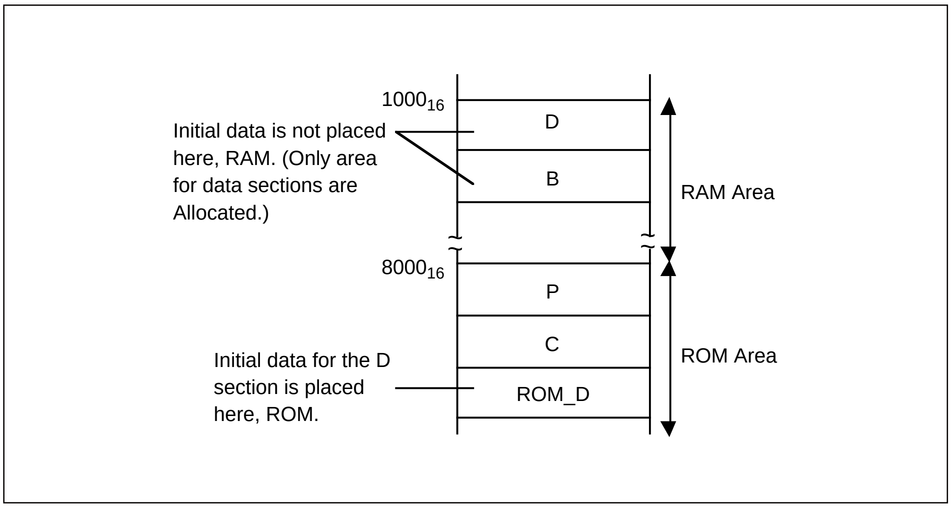

1000_{16}
Initial data is not placed here, RAM. (Only area for data sections are Allocated.)
D
B
RAM Area
8000_{16}
P
C
ROM Area
Initial data for the D section is placed here, ROM.
ROM_D

Figure 7.4 Allocation of Sections for data

Allocation of the section area and data sampling during linking can be specified by the -SEC or -MEM option. For the generated load module, the labels indicating the start and end addresses, respectively, will be automatically generated. These labels are named as shown in Table 7.2.

Table 7.2 Reserved Label Names for Sections

Address	Label Name	Example (D section)	Example (ROM_D section)
Satrt address of section	_ _TOP_	_ _TOP_D	_ _TOP_ROM_D
End address of section	_ _END_	_ _END_D	_ _END_ROM_D

||||| Note |||||

For further information on section allocation and section initialization, refer to the “M3T-CC32R User’s Manual <Assembler>”, “Part 1 Linker lnk32R“. The link can be specified by the command option during start-up of the C compiler.

- Initialization by the start-up program
  - For initialization purposes, the data in the ROM_D section in the ROM area is transferred to the D section of the RAM area.
  - The B section in the RAM area is cleared to zero (filling with 0 data).

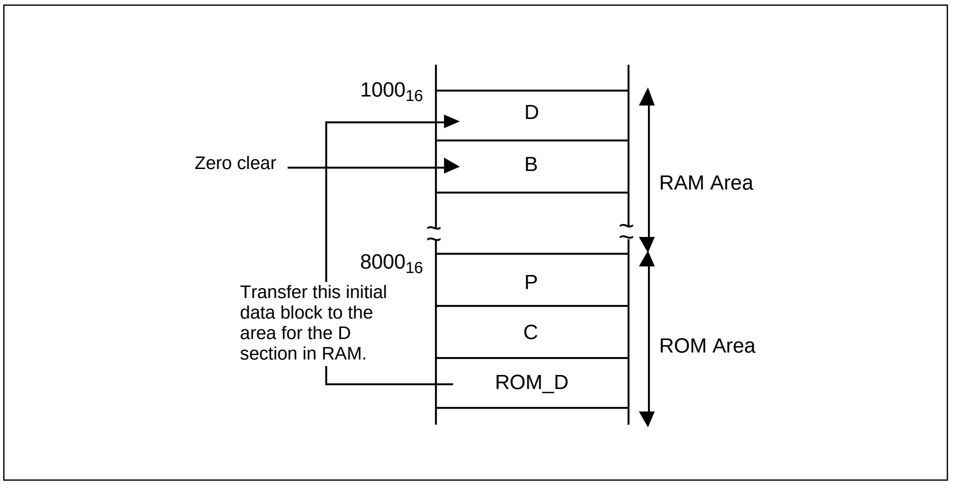

1000₁₆
D
B
Zero clear
RAM Area
8000₁₆
P
C
ROM_D
Transfer this initial data block to the area for the D section in RAM.
ROM Area

Figure 7.5 Initial Setting of Data Area

When programming these processes, use the reserved labels (see Table 7.2) and the section names for referencing the start address of the sections.

7.3.6 Calling the Main Function

By calling the “$main” function, start a user program.
To use the C standarf library in an embedded application, by calling the “$_c_main” function, start a user program.
Initialize the “$_c_main” function with Cstandard library, and call “$main” function.

7.3.7 Start-up Program Example

The following shows an example of the start-up program.

- Start-up program start.ms
  In this sample program, the basic processes, (1) to (5), are performed by the assembly program starting with the “STARTUP”.

```
;; RENESAS TECHNOLOGY CORPORATION AND RENESAS SOLUTIONS CORPO-
RATION

;;
;; start.ms   (2000/06/28)
;;
;;    [ Contents ]
;;
;;        (1) Sample startup routine.
;;        (2) Sample low-level routine.
;;        (3) Heap and stack memory area
;;        (4) Reset vector area
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; (1) Startup routine
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .export         STARTUP
                .export         HALT
                .import         $_c_main
                .section        P,code,align=4
                ;
                ; Initialize PSW control-register.
                ;
STARTUP:        LDI             R0, #128
                MVTC            R0, PSW
                ;
                ; Setting the user and interrupt stack.
                ;
                SETH            R0, #HIGH(U_STACK)
                OR3             R0, R0, #LOW(U_STACK)
                MVTC            R0, SPU
                SETH            R0, #HIGH(S_STACK)
                OR3             R0, R0, #LOW(S_STACK)
                MVTC            R0, SPI
                ;
                ; Clear the B section to zero.
                ;
                SETH            R1, #HIGH(B)
                OR3             R1, R1, #LOW(B)
                SETH            R2, #HIGH(sizeof(B))
                OR3             R2, R2, #LOW(sizeof(B))
                BLEZ            R2, SKIP1
                LDI             R0, #0
LOOP1:          STB             R0, @R1
                ADDI            R1, #1
                ADDI            R2, #-1
                BGTZ            R2, LOOP1
SKIP1:
                ;
                ; Transfer the data in the ROM_D section in ROM
area
                ; to the RAM area.
                ;
                SETH            R1, #HIGH(ROM_D)
                OR3             R1, R1, #LOW(ROM_D)
                SETH            R2, #HIGH(D)
                OR3             R2, R2, #LOW(D)
                SETH            R3, #HIGH(sizeof(ROM_D))
                OR3             R3, R3, #LOW(sizeof(ROM_D))
                BLEZ            R3, SKIP2
LOOP2:          LDB             R0, @R1
                STB             R0, @R2
                ADDI            R1, #1
                ADDI            R2, #1
                ADDI            R3, #-1
                BGTZ            R3, LOOP2
SKIP2:
                ;
                ; Jump to the C standard initialize routine.
                ;
                SETH            R0, #HIGH($_c_main)
                OR3             R0, R0, #LOW($_c_main)
                JL              R0
                ;
                ; End of program ( infinity loop )
                ;
HALT:           BRA             HALT

                .section        C,data,align=4
                .section        ROM_D,data,align=4

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; (2) Sample low-level routine.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .export         $_exit
                .export         $_get_core
                .export         $write
                .export         $_rel_core
                .export         $_strerror
                .export         $close
                .export         $getuniqnum
                .export         $lseek
                .export         $open
                .export         $read
                .export         $getenv
                .export         $raise
                .export         $remove
                .section        P,code,align=4
                ;
                ; Terminate the program run.
                ;
$_exit:         BRA             HALT
                ;
                ; Get the heap memory routine.
                ;
$_get_core:     MV              R1, R4
                SETH            R2, #HIGH(HEAP_POINTER)
                OR3             R2, R2, #LOW(HEAP_POINTER)
                LD              R0, @R2
                ADD             R1, R0
                ST              R1, @R2
                JMP             R14
                ;
                ; Write data to memory buffer routine.
                ;
$write:         LDI             R3, #0
                BRA             SKIP3
LOOP3:          SETH            R1, #HIGH(WRITE_POINTER)
                OR3             R1, R1, #LOW(WRITE_POINTER)
                LD              R2, @R1
                ADD3            R0, R2, #1
                ST              R0, @R1
                MV              R1, R5
                LDI             R0, #H'3FF
                ADD             R1, R3
                REM             R2, R0
                SETH            R0, #HIGH(WRITE_BUFFER)
                OR3             R0, R0, #LOW(WRITE_BUFFER)
                ADDI            R3, #1
                ADD             R2, R0
                LDB             R1, @R1
                STB             R1, @R2
SKIP3:          LD              R0, @R15
                CMP             R3, R0
                BC              LOOP3
                LD              R0, @R15
                JMP             R14
                ;
                ; ( Not implemented routine )
                ;
$_rel_core:
$_strerror:
$close:
$getuniqnum:
$lseek:
$open:
$read:
$getenv:
$raise:
$remove:        LDI             R0, #0
                JMP             R14
                ;
                ; low-level routines use area.
                ;
                .export         WRITE_BUFFER
                .section        D,data,align=4
HEAP_POINTER:   .DATA.W         HEAP
WRITE_POINTER:  .DATA.W         0
WRITE_BUFFER:   .RES.B          H'400
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; (3) Heap and stack memory area ( common use )
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .section        B,data,align=4
HEAP:
                .RES.B          H'4000
U_STACK:
                .RES.B          H'1000
S_STACK:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; (4) Reset vector area
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .section        RI,code,locate=h'7FFFFFF0
                SETH            R0, #HIGH(STARTUP)
                OR3             R0, R0, #LOW(STARTUP)
                JMP             R0

                .end            STARTUP

;; RENESAS TECHNOLOGY CORPORATION AND RENESAS SOLUTIONS CORPO-
RATION
```

❅This “Start-up program start.ms” is sample program.

7.4 About start-up file start.ms in HEW

The Hew generates a file "start.ms" when creating new project. This file was modified from one that was using with the TM and the User's Manual.

Fundamentally, the contents of these start.ms are nearly equal. However, the start.ms HEW generated can be controled by the assembler as32R with setting following paramter into -D option. If you will modify this start.ms, be careful in this point.

Table 7.3. Meaning of control symbols of start.ms the HEW generated

SymbolName	Item name of HEW project Name creating dialog	Means	Initial value
__STANDARD_IO__	Use Standard I/O Library	Decides whether or not that the start.ms initializes the standard library before calling function main().	0 (The standard library is not initialized.
__HEAPSIZE__	Heap Size	Heap Size (for malloc(), etc.)	H'4000
__USTACKSIZE__	User Stack PointerStack Size	User Stack (SPU pointed) Size	H'1000
__ISTACKSIZE__	Interrupt Stack PointerStack Size	Interrupt Stack (SPI pointed) Size	H'1000

7.5 In-line Assembling

7.5.1 Overview of In-line Assembling

Embedded application also requires low level processes such as controlling of hardware and operating system (OS) as well as high speed efficient functional processes. To meet these various processing requirements, the C compiler supports the "In-line assembly function" which allows to write an assembly language command in the C language source program.

To insert an assembly description into the C source program, use the special function asm. The asm function allows embedding an assembly language code into the C source program.

||||| Note |||||

The C compiler optimization feature is powerful when used for removing unnecessary codes, replacing commands and substituting commands. When using the in-line assembling feature, take into consideration effects of C compiler optimization feature.

7.5.2 How to Write the asm Function

To use the asm function, use the following definition.

```
#pragma keyword asm on
```

This does not specify the "asm" as a normal identifier but as a reserved word. To use the asm again as the identifier, write the following :

```
#pragma keyword asm off
```

Write the asm function in the format shown in Figure 7.6.

```
#pragma keyword asm on
asm ( "assembly_code"[,argument1] [,argument2] );
```

assembly_code	: The assembly code to be embedded
argument1	: Expression to be set to register R0
argument2	: Expression to be set to register R1

Figure 7.6 Format of asm function

Write the assembly language code to be embedded into the *assembly_code* field by following the assembler writing specification. An escape sequence as well as a string literal (see 4.1.4) can be written. Since the beginning of an *assembly_code* is recognized as a label, the first mnemonic must be preceded by one or more space characters.

Example :

```
asm("  ldi R0, #h'10");
```

↑ one or more spaces

With the asm function, up to 2 arguments (expressions) can be specified in addition to assembly code. Each argument is evaluated before executing the assembly code of the asm function. A value stored in *argument1* is set to register R0 and that stored in *argument*2 to register R1. A valid argument value is an integer only. (Bear in mind that integral promotion described in 4.3.2 “Implicit Conversions” will be applied to signed and unsigned, char and short types.) Arguments having other types may not guarantee proper C compiler operation.

Example :

```
int  i, j;
asm ( "  add R0, R1", i, j );
```

The variable i is preset to R0, and the variable j preset to R1.

If no argument is specified, the contents of R0 and R1 are unknown.

7.5.3 Limitations of asm Function

When writing the asm function, take the following limits into consideration :

- Limited register usage

  The asm function can normally recognize four registers R0-R3. When using the other registers and the accumulator in the asm function, the user program must assure that the contents of the register are recovered after the function releases the register. To do so, save the contents of the register before the asm function uses it and then return the contents to the register upon releasing of the register by the function.

  Example : Saving and recovering of register R4 within asm function

```
asm(" ST R4,@-sp\n"    /* Save R4 */
    " ........... \n"  /* Any process
                          using R4 */
    " LD R4,@sp+\n");  /* Restore R4 */
```

- Limits during compiling (optimization specified)

  During optimization, the compiler deletes unnecessary codes and replaces and substitutes instructions. When compiling and specifying optimization of the source file using the asm function, take into consideration the effects of optimization by the C compiler.

- Error check considerations

  The C compiler does not check the contents of the assembly code used in the asm function. If the asm function contains invalid assembly code, the error is detected by the assembler and the error message is output to the assembly source file, but not to the C source file. If an assembler error message is output after starting the C compiler, first check the contents of the asm function.

- Limitations on parameter specification

  Although expressions can be written in asm function parameters, we recommend that only constant expressions and identifiers, but no other expressions, be written. If an expression like the one shown below is written, the C compiler may not operate correctly.

  · An expression that has side effect
  (e.g., operators such as ++, -, =, +, -; expressions that contain a function call)
  · Complicated expression

- Limitations on length of asm function

  Up to about 1,000 characters can be written in an asm function. When writing multiple lines of assembly code, we recommend that they be divided and written in multiple asm functions.

- Limitations on labels

  We recommend using labels which are not the duplicates of those generated internally by the compiler (label names beginning with the underscore '_'). Make sure any parts of label names except the underscore are not the duplicates of symbol or function names defined in C language.

- Limitations on instructions writable in asm function

  Pseudo-instruction and macro-instructions cannot be written in a asm function.

- Limitations on optimization of a program in which asm functions are written

  When a program that contains a description of one or more asm functions is compiled after specifying optimization, a warning message like the one shown below may appear, with the optimization partly suppressed. We recommend that functions which use the inline assembly facility be defined in another module.

· Warning message

```
<command line>:warning: xxx.c: unable to optimize -- skipped phase
```

- Other limits

The following descriptions should not be included in the asm function. If such a description is included in the asm function, the user must be responsible for the results**.**

- Branch instructions
- Assembler's pseudo-instructions and macro-instructions
- Changing contents of stack
- Label definitions
- Reference to C compiler-generated labels

7.5.4 asm Function Example

Figure 7.7 shows examples using the asm functions.

```
/* Multiply arrays X[cnt] and Y[cnt], and obtain the result */
#pragma keyword asm on
void sumXY(short *X, short *Y, int cnt, int *output)
{
    asm("       mvtachi r0\n"
        "       mvtaclo r0”, 0);
    for ( ; cnt-- > 0; ++X, ++Y )
        asm("   macwlo  r0, r1", *X, *Y);
    asm("       mvfachi r3\n"
        "       st r3, @r0\n"
        "       mvfaclo r3\n"
        "       st r3, @+r0\n", (int)output);
}
```

Figure 7.7 Example of asm Functions

Figure 7.8 shows the compiled source program shown in Figure 7.7.

```
    .IMPORT $_100_builtin_memcopy
    .SECTION    P,CODE,ALIGN=4
    .EXPORT $sumXY
$sumXY:
    LDI     R0,#0
    mvtachi r0
    mvtaclo r0
    BRA     L5
L4:
    LDH     R0,@R4
    LDH     R1,@R5
    macwlo  r0, r1
L2:
    ADDI    R4,#2
    ADDI    R5,#2
L5:
    MV      R1,R6
    ADD3    R6,R1,#-1
    BGTZ    R1,L4
L3:
    MV      R0,R7
    mvfachi r3
    st      r3,@r0
    mvfaclo r3
    st      r3,@+r0

L1:
    JMP     R14
    .END
```

Figure 7.8 Example of Compiled asm Functions

Chapter 8

Standard Header Files

8.1 Overview of the Header Files

A standard header file is a file in which prototype declaration, macro definition and data type declaration necessary to for use of the C standard libraries are written. Available standard header files include 15 types (Table 8.1). When using a C standard library function, the header file containing definition and declaration required to execute the library function must be included for each process.

Table 8.1 shows the standard header files and associated library functions (type name).

Table 8.1 Standard Header Files

Header File	Description	Associated Library Function
assert.h	Macro definition that outputs program diagnostic information	Program diagnostic function
ctype.h	Macro definition of the character handling function and character check function	Character handling function
errno.h	Macro definition related to the error number	All functions (as necessary)
float.h	Macro definition of the limit value related to internal representation of a floating-point number	Mathematics function,etc. (only when float.h macro is used)
limits.h	Macro definition of the limit value related to the compiler internal process	All functions(as necessary)
locale.h	Declaration of the locale (localization) handle function	Localization function
math.h	Declaration of the double and float type mathematical unction and macro definition	Mathematics function
mathf.h	Declaration of the float type mathematical function and macro definition	Mathematics function
setjmp.h	Declaration of the branch function, data type declaration	Non-local jump function
signal.h	Signal (interrupt) declaration of the number of processes	Signal handling function
stdarg.h	Macro declaration of variable arguments functions, Data type declaration	Variable arguments access function
stddef.h	Definition common to standard headers, data type declaration	All functions (as necessary)
stdio.h	Declaration of the input and output functions, data type declaration, macro definition	Input /output function
stdlib.h	Declaration of the C program standard process function, e.g., memory management, data type declaration, macro definition	General utility function
string.h	Declaration of the string handle function and memory handle function	String handling function
time.h	Declaration of the date and time handle functions	Date and time function

8.2 Contents of the Header Files

The section describes standard library functions declared or defined in standard header files. To use these functions, the associated header file must be included. The macros defining limit values are listed. The header files are listed in the alphabetical order (8.2.1-8.2.15).

8.2.1 assert.h

Defines the program diagnostic function assert (macro definition with parameter).

8.2.2 ctype.h

Prototype declaration and macro definition of the character handling function. These functions are listed in Table 8.2.

Table 8.2 Functions Declared by ctype.h

Function	Description	Reentrant
isalnum	Judges whether a letter or decimal digit.	❍
isalpha	Judges whether a letter or not.	❍
iscntrl	Judges whether a control character or not.	❍
isdigit	Judges whether a decimal digit or not.	❍
isgraph	Judges whether a printable character other than space.	❍
islower	Judges whether a lower case letter or not.	❍
isprint	Judges whether a printable character including space.	❍
ispunct	Judges whether a special character or not.	❍
isspace	Judges whether a white-space or not.	❍
isupper	Judges whether an upper case letter or not.	❍
isxdigit	Judges whether a hexadecimal digit or not.	❍
tolower	Converts an upper case letter into lower case.	❍
toupper	Converts a lower case letter into upper case.	❍

8.2.3 errno.h

Upon occurrence of an error, defines the external variable errno that holds the error number and the macro that indicates the error number. Table 8.3 shows the macros defined by errno.h.

Table 8.3 Macros Defined by errno.h

Macro Name	Description
EDOM	If the value of input parameter is outside the range defined by a function, EDOM indicates the value to be set to errno.
ERANGE	If the calculation of a function results in a value which cannot be expressed in double type, or if it results in an overflow or underflow, ERANGE indicates the value to be set to errno.

8.2.4 float.h

Defines the limits concerning internal representation of a floating-point number. Table 8.4 describes the macros defined by float.h.

Table 8.4 Macros Defined by float.h (1/3)

Macro Name	Description
FLT_RADIX	Indicates the radix of an exponent.
FLT_ROUNDS	Indicates the round mode in addition operation. The macro definitions are as shown below : • Round to the nearest value : 1 • Round toward +∞ : 2 • Round toward -∞ : 3 • Round down to 0 : 0 • Not specified : -1 Round and round off are process system definitions.
FLT_MAX	Indicates the max. positive value which can be expressed in a float type floating-point number.
DBL_MAX	Indicates the max. positive value which can be expressed in a double type floating-point number.
LDBL_MAX	Indicates the max. positive value which can be expressed in a long double type floating-point number.
FLT_MIN	Indicates the min. positive value which can be expressed in a float type floating-point number.

Table 8.4 Macros Defined by float.h (2/3)

Macro Name	Description
DBL_MIN	Indicates the min. positive value which can be expressed in a double type floating-point number.
LDBL_MIN	Indicates the min. positive value which can be expressed in a long double type floating-point number.
FLT_MAX_EXP	Indicates the max. value of the power of the base which can be expressed in a float type floating-point number.
DBL_MAX_EXP	Indicates the max. value of the power of the base which can be expressed in a double type floating-point number.
LDBL_MAX_EXP	Indicates the max. value of the power of the base which can be expressed in a long double type floating-point number.
FLT_MIN_EXP	Indicates the min. value of the power of the base which can be expressed in a float type floating-point number.
DBL_MIN_EXP	Indicates the min. value of the power of the base which can be expressed in a double type floating-point number.
LDBL_MIN_EXP	Indicates the min. value of the power of the base which can be expressed in a long double type floating-point number.
FLT_MAX_10_EXP	Indicates the max. value of the power of 10 which can be expressed in a float type floating-point number.
DBL_MAX_10_EXP	Indicates the max. value of the power of 10 which can be expressed in a double type floating-point number.
LDBL_MAX_10_EXP	Indicates the max. value of the power of 10 which can be expressed in a long double type floating-point number.
FLT_MIN_10_EXP	Indicates the min. value of the power of 10 which can be expressed in a float type floating-point number.
DBL_MIN_10_EXP	Indicates the min. value of the power of 10 which can be expressed in a double type floating-point number.
LDBL_MIN_10_EXP	Indicates the min. value of the power of 10 which can be expressed in a long double type floating-point number.
FLT_DIG	Indicates the max. No. of digits of decimal precision of a float type floating-point number.
DBL_DIG	Indicates the max. No. of digits of decimal precision of a double type floating-point number.

Table 8.4 Macros Defined by float.h (3/3)

Macro Name	Description
LDBL_DIG	Indicates the max. No. of digits of a decimal precision of a long double type floating-point number.
FLT_MANT_DIG	Indicates the max. No. of digits of the mantissa of a float type floating-point number (when expressed to radix).
DBL_MANT_DIG	Indicates the max. No. of digits of the mantissa of a double type floating-point number (when expressed to radix).
LDBL_MANT_DIG	Indicates the max. No. of digits of the mantissa of a long double type floating-point number (when expressed to radix).
FLT_EPSILON	Indicates the min. floating-point number *x*, 1.0 + *x* ≠ 1.0 of the float type.
DBL_EPSILON	Indicates the min. floating-point number *x*, 1.0 + *x* ≠ 1.0 of the double type.
LDBL_EPSILON	Indicates the min. floating-point number *x*, 1.0 + *x* ≠ 1.0 of the long double type.

Classification	definition name	value	Explanation
	FLT_GUARD	1	Specifies whether or not the multiplication calculation result. The macro definitions are as shown below : *Use the guard-bit : 1 *Not use the guard-bit : 0
	FLT_NORMALIZE	1	Specifies whether or not the library normalize floating numeric number.The macro definitions are as shown below : *Normalize : 1 *Not normalize : 0
	FLT_EXP_DIG	8	Indicates the max.No.of digits of binary precision of the power of the base which can be expressed in a float type floating-point number.
	DBL_EXP_DIG	11	Indicates the max.No.of digits of binary precision of the power of the base which can be expressed in a double type floating-point number.

Classification	definition name	value	Explanation
	LDBL_EXP_DIG	11	Indicates the max.No.of digits of binary precision of the power of the base which can be expressed in a long double type floating-point number.
	FLT_POS_EPS	5.9604648328104304e-8F	Indicates the min.floating-point number x ,1.0 +x ≠1.0 of the float type.
	DBL_POS_EPS	1.1102230246251567e-16	Indicates the min.floating-point number x ,1.0 +x ≠1.0 of the double type.
	LDBL_POS_EPS	1.1102230246251567e-16	Indicates the min.floating-point number x ,1.0 +x ≠1.0 of the long double type.
	FLT_NEG_EPS	5.9604642999033786e-8F	Indicates the min.floating-point number x ,1.0 -x ≠1.0 of the float type.
	DBL_NEG_EPS	1.1102230246251565e-16	Indicates the min.floating-point number x ,1.0 -x ≠1.0 of the double type.
	LDBL_NEG_EPS	1.1102230246251565e-16	Indicates the min.floating-point number x ,1.0 -x ≠1.0 of the long double type.
	FLT_POS_EPS_EXP	-23	Indicates the min.No.of the power of the binary base which can be expressed floating-point number x,1.0 +x ≠1.0 of the float type.
	DBL_POS_EPS_EXP	-52	Indicates the min.No.of the power of the binary base which can be expressed floating-point number x,1.0 +x ≠1.0 of the double type.
	LDBL_POS_EPS_EXP	-52	Indicates the min.No.of the power of the binary base which can be expressed floating-point number x,1.0 +x ≠1.0 of the long double type.

Classification	definition name	value	Explanation
Macro	FLT_NEG_EPS_EXP	-24	Indicates the min.No.of the power of the binary base which can be expressed floating-point number x,1.0 -x ≠1.0 of the float type.
Macro	DBL_NEG_EPS_EXP	-53	Indicates the min.No.of the power of the binary base which can be expressed floating-point number x,1.0 -x ≠1.0 of the double type.
Macro	LDBL_NEG_EPS_EXP	-53	Indicates the min.No.of the power of the binary base which can be expressed floating-point number x,1.0 -x ≠1.0 of the long double type.

8.2.5 limits.h

Defines limits concerning numerical value of each type. Table 8.5 shows the macros defined.

Table 8.5 Macros Defined by Limits.h

Macro Name	Description
CHAR_BIT	Indicates the number of bits composing a char type.
CHAR_MAX	Indicates the max. value that a char type variable can have.
CHAR_MIN	Indicates the min. value that a char type variable can have.
SCHAR_MAX	Indicates the max. value that a signed char type variable can have.
SCHAR_MIN	Indicates the min. value that a signed char type variable can have.
UCHAR_MAX	Indicates the max. value that an unsigned char type variable can have.
SHRT_MAX	Indicates the max. value that a short int type variable can have.
SHRT_MIN	Indicates the min. value that a short int type variable can have.
USHRT_MAX	Indicates the max. value that an unsigned short int type variable can have.
INT_MAX	Indicates the max. value that an int type variable can have.
INT_MIN	Indicates the min. value that an int type variable can have.
UINT_MAX	Indicates the max. value that an unsigned int type variable can have.
LONG_MAX	Indicates the max. value that a long type variable can have.
LONG_MIN	Indicates the min. value that a long type variable can have.
ULONG_MAX	Indicates the max. value that an unsigned long type variable can have.

8.2.6 locale.h

Performs prototype declaration and macro definition of a function (locale handle function) which processes localization of the program. Table 8.6 and Table 8.7 list the functions declared and macros declared by locale.h.

Table 8.6 Functions Declared by locale.h

Function	Description	Reentrant
localeconv	Initialize struct lconv.	×
setlocale	Sets and searches for locale information.	×

Table 8.7 Macros Defined by locale.h

Macro Name	Description
LC_ALL	Sets and searches all locale information.
LC_COLLATE	Sets and searches information that affects the strcoll function and strxfrm function.
LC_CTYPE	Sets and searches information that affects functions handling character and multi bytes, except for the isdigit and isxdigit functions.
LC_MONETARY	Sets and searches information that affects currency information returned from the localeconv function.
LC_NUMERIC	Sets and searches information that affects information other than decimal point used by the input and output functions and the character handling functions and currency information returned from the localeconv function.
LC_TIME	Sets and searches information that affects the strftime function.

8.2.7 math.h

Performs prototype declaration and macro definition of arithmetic functions. Table 8.8 and Table 8.9 list the functions declared and macros declared by math.h.

Table 8.8 Functions Declared by math.h

Function	Description	Reentrant
acos	Obtains the arc cosine of a floating-point number.	×
asin	Obtains the arc sine of a floating-point number.	×
atan	Obtains the arc tangent of a floating-point number.	×
atan2	Divides a floating-point number by a floating-point number and obtains the arc tangent of the result.	×
ceil	Computes the integer ceiling of a floating-point number.	×
cos	Obtains the cosine of radians of a floating-point number.	×
cosh	Obtains the hyperbolic cosine of a floating-point number.	×
exp	Obtains the exponential function of a floating-point number.	×
fabs	Obtains the absolute value of a floating-point number.	×
floor	Cuts off the fraction of a floating-point number.	×
fmod	Computes the floating-point remainder.	×
frexp	Divides a floating-point number into products of value (0.5, 1.0) and 2 to the *n*th power.	×
ldexp	Performs multiplication of a floating-point number and 2 to the *n*th power.	×
log	Obtains natural logarithm of a floating-point number.	×
log10	Obtains the base 10 logarithm of a floating-point number.	×
modf	Divides a floating-point number into integer and fractional parts.	×
pow	Obtains a floating-point number to *n*th power.	×
sin	Obtains the sine of the radians of a floating-point number.	×
sinh	Obtains hyperbolic sine of a floating-point number.	×
sqrt	Obtains the positive square root of a floating-point number.	×
tan	Obtains the tangent of the radians of a floating-point number.	×
tanh	Obtains hyperbolic tangent of a floating-point number.	×

Table 8.9 Macro Defined by math.h

Macro Name	Description
HUGE_VAL	Indicates the value that the function returns when its operation results in an overflow.

8.2.8 setjmp.h

Performs the prototype declaration of a branch function and necessary data type declaration. The declared functions and data types are shown in Table 8.10 and Table 8.11.

Table 8.10 Functions Declared by setjmp.h

Function	Description	Reentrant
longjmp	Recovers the execution environment saved by setjmp and transfers control to the program location of a setjmp call.	❍
setjmp	Saves the current environment to a memory area.	❍

Table 8.11 Data Types Declared by setjmp.h

Data Type	Description
jmp_buf	Indicates the type name corresponding to the memory location where information that enables movement of the control between functions.

8.2.9 signal.h

Performs the prototype declaration and macro definition of the signal process functions. The declared functions are shown in Table 8.12.

Table 8.12 Functions declared by signal.h

Function	Description
raise	Send a signal to the executing-program.
signal	Sets up a signal handler that responds to the signal.

8.2.10 stdarg.h

Performs the macro definition and data type declaration required for the variable arguments functions. The defined macros and declared data types are shown in Table 8.13 and Table 8.14. Theses macros are further described in 9.2 "Library Function Descriptions".

Table 8.13 Macros Defined by stdarg.h

Macro Name	Description	Reentrant
va_arg (Macro)	Gets variable arguments in turn.	❍
va_end (Macro)	Ends the reference to variable arguments.	❍
va_start (Macro)	Initializes to reference variable arguments.	❍

Table 8.14 Data Types Declared by stdarg.h

Data Type	Description
va_list	Indicates the types of variable arguments used by the macros va_start, va_arg and va_end for referencing variable arguments.

8.2.11 stddef.h

Defines the macros which are commonly used by the standard header files and declares the common data types. The defined macros and declared data types are shown in Table 8.15 and Table 8.16.

Table 8.15 Macros Defined by stddef.h

Macro Name	Description
errno	Every time an error occurs while processing the errno library function, the error code set for that library is set to the errno macro. To check the error that occurred during processing of a library function, set the errno to 0 before calling the library function and then check the code set into errno after the process.
NULL	Indicates the value when the pointer is pointing to nothing.

Table 8.16 Data Types Declared by stddef.h

Data Type	Description
ptrdiff_t	Indicates the type resulting from subtraction of two pointers.
size_t	Indicates the type resulting from operation of the sizeof operator.
wchar_t	Indicates the type of the wide-character (`L'...'`)

8.2.12 stdio.h

Performs the prototype declaration of the input and output functions and the definition of macro definition and data type declaration required for input and output functions. The declared functions, defined macros and data types are shown in Table 8.17, Table 8.18 and Table 8.19, respectively.

Table 8.17 Functions Declared by stdio.h (1/2)

Function	Description	Reentrant
clearerr	Clears an error condition in a stream.	×
fclose	Closes a file.	×
feof	Checks if the end of a stream is reached.	×
ferror	Checks if a stream is in an error condition.	×
fflush	Outputs the contents of a stream to a file.	×
fgetc	Gets a character from a stream.	×
fgetpos	Locates the current position on a stream.	×
fgets	Gets a string from an input stream.	×
fopen	Opens a file.	×
fprintf	Outputs data to a stream according to the format.	×
fputc	Outputs a character to a stream.	×
fputs	Outputs a string to a stream.	×
fread	Transfers data from a stream to a memory area.	×
freopen	Closes a currently opened stream, and reopens a new file with the new file name.	×
fscanf	Gets data from a stream, and converts the data by following the format.	×
fseek	Moves the current read/write position within a stream.	×
fsetpos	Changes the current position on a stream.	×
ftell	Locates the current read/write position in a stream.	×
fwrite	Transfers data from a memory area to a stream.	×
getc	Gets one character from a stream.	×
getchar	Gets a character from the standard input (stdio).	×
gets	Gets a string from the standard input (stdio).	×
perror	Outputs the error message corresponding to the error code to the standard error file (stderr).	×
printf	Converts data by following the format and outputs it to the standard output file (stdout).	×

Table 8.17 Functions Declared by stdio.h (2/2)

Function	Description	Reentrant
putc	Outputs a character to a stream.	×
putchar	Outputs a character to the standard output file (stdout).	×
puts	Outputs a string to the standard output file (stdout).	×
remove	Deletes a file.	Depends on user description
rename	Renames a file.	Depends on user description
rewind	Moves the current read/write position on a stream to the beginning of the file.	×
scanf	Gets data from the standard input file (stdin) and converts the data by following the format.	×
setbuf	Defines a buffer for an I/O stream.	×
setvbuf	Defines and sets a buffer for an I/O stream.	×
sprintf	Converts the data by following the format and outputs the data to an area.	×
sscanf	Gets data from a memory area and converts the data by following the format.	×
tmpfile	Creates a temporary file.	×
tmpnam	Creates a not-existing temporary file name.	×
ungetc	Returns a character a stream.	×
vfprintf	Outputs a variable argument list to a stream by following the format.	×
vprintf	Outputs a variable argument list to the standard output (stdout) by following the format.	×
vsprintf	Outputs a variable arguments list to a memory area by following the format.	×

Table 8.18 Macros Defined by stdio.h

Macro Name	Description
_IOFBF	Indicates that all input and output processes use the buffer area.
_IOLBF	Indicates that input and output processes use the buffer area in units of one line.
_IONBF	Indicates that input and output processes do not use the buffer area.
BUFSIZ	Indicates the size of the buffer required for input and output processes.
EOF	End of file, i.e., indicates that no further input will come from the file.
L_tmpnam	Indicates that the size of the location is enough to store the string of the name of the temporary file generated by the tmpnam function.
SEEK_CUR	Indicates that the current read/write position of the file is moved from the current of file to the offset.
SEEK_END	Indicates that the current read/write position of the file is moved from the end of the file to the offset.
SEEK_SET	Indicates that the current read/write position of the file is moved from the beginning of the file to the offset.
TMP_MAX	Indicates the maximum number of file names generated by the tmpnam function.
FOPEN_MAX	Indicates the maximum number of files including stdio, stdout and stderr, that the fopen function can open at the same time.
FILENAME_MAX	Indicates the maximum size of the name of the file that can be opened.
stderr	The file pointer which indicates the standard error.
stdin	The file pointer which indicates the standard input file.
stdout	The file pointer which indicates the standard output file.

Table 8.19 Data Type Declared by stdio.h

Data Type	Description
FILE	Indicates the type of structure that stores the pointer of the buffer necessary for the stream input and output processes and error indicator and end indicator and other control information.
fpos_t	Used by fsetpos and fgetpos functions to indicate file position.

8.2.13 stdlib.h

Performs the prototype declaration of general utility functions (memory management, end process), definition of macros, and declaration of data types necessary for general utility functions. These functions, macros and data types are shown in Tables 8.20, 8.21 and 8.22.

Table 8.20 Functions Declared by stdlib.h(1/2)

Function	Description	Reentrant
abort	Puts the running program to forced stop.	×
abs	Obtains the absolute value of an int type integer.	❍
atexit	Catalogs the function to be called upon successful termination of the program.	×
atof	Converts the character string representing a number into a double type floating-point number.	×
atoi	Converts the character string representing a decimal number into a int type integer.	×
atol	Converts the character string representing a decimal number into a long type integer.	×
bsearch	Performs binary search.	❍ (Depends on comparison function)
calloc	Allocates a memory space and initializes the allocated memory space to 0.	×
div	Divides an int type integer and obtains the quotient and remainder.	❍
exit	Terminates the program.	×
free	Releases the specified memory area.	×
getenv	Gets the content of an environmental variable.	Depends on user description
labs	Obtains the absolute value of a long type integer.	❍
ldiv	Divides a long type integer and obtains the quotient and remainder.	❍
malloc	Allocates memory area.	×
mblen	Obtains the number of bytes composed of multibyte characters.	❍
mbstowcs	Converts a multibyte character string into a wide character string.	❍
mbtowc	Converts a multibyte character into a wide character.	❍
qsort	Performs sorting.	❍ (Depends on comparison function)

Table 8.20 Functions Declared by stdlib.h(2/2)

Function	Description	Reentrant
rand	Generates a pseudo-random integer which resides between 0 and RAND_MAX.	×
realloc	Changes the memory area size to the specified size.	×
srand	Sets the initial values of the pseudo-random integers which the rand function generates.	×
strtod	Converts a string into a double type floating-point number.	×
strtol	Converts a string into a long type integer.	×
strtoul	Converts a string into an unsigned long type integer.	×
system	Passes a command string to the host environment.	Depends on user description
wcstombs	Converts a wide string into a multibyte string.	❍
wctomb	Converts a wide character into a multibyte character.	❍

Table 8.21 Macro Defined by stdlib.h

Macro Name	Description
RAND_MAX	Indicates the maximum value of a pseudo-random integer generated by the rand function.

Table 8.22 Data Type Declared by stdlib.h

Data Type	Description
div_t	Indicates the type of structure of the value div function returns.
ldiv_t	Indicates the type of structure of the value ldiv function returns.

8.2.14 string.h

Performs the prototype declaration of the character string handle function and memory handle function. Table 8.23 shows the functions declared by string.h.

Table 8.23 Functions Declared by string.h

Function	Description	Reentrant
memchr	Locates, in a memory area, the position where a character first appears.	❍
memcmp	Compares the contents of two memory areas.	❍
memcpy	Copies the contents of a memory area to the destination memory area.	❍
memmove	Moves the contents from a memory area to the destination memory area.	❍
memset	Copies a character into the first n characters in memory area.	❍
strcat	Links a string to the end of a string.	❍
strchr	Locates, in a string, the position where a character first appears.	❍
strcmp	Compares two strings.	❍
strcoll	Compares the two strings based on the current locale.	❍
strcpy	Copies the contents (including null characters) of the source string to the target memory area.	❍
strcspn	Computes the length of initial segment of a string which consists of unspecified characters.	❍
strerror	Returns the error message.	❍ (Depends on _strerror function)
strlen	Measures the size of string.	❍
strncat	Links the specified number of characters to the string.	❍
strncmp	Compares specified number of characters of two strings.	❍
strncpy	Copies the specified number of characters from the string to memory.	❍
strpbrk	Locates the position where one of the specified characters first appears in a string.	❍
strrchr	Locates the position where a character last appears in a string.	❍
strspn	Computes the length of initial segment of a string which consists of specified characters.	❍
strstr	Finds the first occurrence point of a string within another.	❍
strtok	Separates a string into tokens.	✕
strxfrm	Converts the string based on the current locale.	❍

8.2.15 time.h

Performs the prototype declaration and macro definition of the date and time handle functions. Table 8.24 shows the declared functions.

Table 8.24 Functions Declared by time.h

Function	Description	Reentrant
asctime	Converts data and time (a struct tm) into the equivalent text string.	×
clock	Gets the elapsed processor time.	Depends on user description
ctime	Converts the calendar time (a time_t value) into the equivalent text string.	×
difftime	Computes the difference between the two specified times.	❍
gmtime	Converts calendar time to Coordinated Universal Time (UTC).	×
localtime	Converts current calendar time to the local time.	×
mktime	Converts date and time (a struct tm) to the calendar time.	×
strftime	Converts date and time (a struct tm) to the format specified.	×
time	Reads the current calendar time.	Depends on user description

Chapter 9

C Standard Library

This Chapter describes the C standard libraries contained in CC32R.

9.1 Overview of the C Standard Library

9.1.1 The Library Files Contained in CC32R

CC32R provides two versions of the library which should be used properly according to the argument passing method for library functions : "m32RcR.lib, m32RcRM.lib, m32RcRL.lib" for "register- passing".

- m32RcR.lib, m32RcRM.lib, m32RcRL.lib

Specify either libraries with the -l option as you invoke the C compiler or linker.

||||| Note |||||

Do not mix-use m32Rc.lib, m32RcM.lib, m32RcL.lib (for stack-passing) and m32RcR.lib, m32RcRM.lib, m32RcRL.lib (for register-passing) in an application.

9.1.2 Library Function Groups

C standard libraries are in conformity to ANSI specifications (ANSI/ISO 9899-1990). The functions are classified into 11 groups as shown in Table 9.1 Each library can be made available for use in a unit process only when the corresponding standard header file is included.

Table 9.1 C Standard Library Function Groups

Group	Summary	Corresponding Header file
Program diagnostic function	Program diagnostic information output	assert.h
Character handling function	Character handling and checking	ctype.h
Mathematics function	Arithmetic operations e.g., trigonometric functions	math.h
Non-local jump function	Transfer of control from function to function	setjmp.h
Variable arguments access function	Access to arguments of variable arguments function	stdarg.h
Input/output function	Input and output operations	stdio.h
General utility function	C program standard processes, e.g., memory management	stdlib.h
String handling function	Comparison, copy, etc., of character string	string.h
Localization function	Setting and handling of locale (localization)	locale.h
Date and time function	Setting, changing, etc. of date and time	time.h
Signal handling function	Transferring of signal (interrupt)	signal.h

In addition, there are “initialization function” and “termination function” that effect several settings for using functions shown above. They are undefined in ANSI-C. Their function names begin with an underscore (_) . In general, they should be used in the start-up program (see 7.3.6 to 7.3.8).

9.1.3 Consideration for using the Library

When using C standard library, observe the following precautions :

- Include the standard header files.

  If you use the C standard library, include necessary standard header files. For header files, refer to Chapter 8 "Standard Header Files". Header file(s) for each function are described in Syntax in 9.2 "Library Function Descriptions" .

- Do not create any function of the same name as the C standard library function.

  In a user program, do not create any function named as the same name as one of C standard library functions, which conforms to the ANSI-C specifications. It is not recommendable.

9.1.4 Library Error Message

Some library function sets an error number in `errno` defined in the standard header file errno.h, when an error occurs during execution of the library function. An error message is defined by the error number and such an error message can be output. An example of program to check `errno` is shown below :

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <string.h>
void main(void)
{
  double x,a=2.0;

  x = asin(a);
  if( errno == EDOM )
   printf( "%s\n",strerror(errno) ); /* print error message */
}
```

Figure 9.1 Example of Checking errno

In the example in Figure 9.1, an arc sine value of a number is calculated through the asin function. When an argument a exceeds the domain [-1, 1] of the asin function, a value EDOM is set in `errno` and, therefore, an error message is output by the printf function. When an error number is passed as a real argument, the strerror function returns a string pointer to the corresponding error message.

9.2 Rebuild to Method of Standard Library

The source environment of the standard library is attached.
Using this environment, the library can be created using any code generation option.Explain the procedure of creating a library using this environment.
This description is for the MS-Windows(PC) version. If you are using the EWS version, change the following as shown:

MS-Windows(PC) version		EWS version
lib32\src	⇒	lib/src
BUILD.bat	⇒	BUILD.csh
CLEAN.bat	⇒	CLEAN.csh

9.2.1 Library Building Procedure

<Storage location>
When CC32R is installed, directory name lib32R\src is created in the installation directory and the required directories and files created in that directory.

(1) Modify the BUILD batch file.
Specify the library name and compile options (code generation-related items; memory model, and specification of optimization). Change the following in BUILD.bat.

```
    set TARGET=<TARGET NAME>
    set CFLAGS=<COMPILE OPTION>
    set AFLAGS=<ASSEMBLE OPTION>
    set MMODEL=<MEMORY MODEL>
```

■<TARGET NAME>
Specify the filename (excluding extension) of the library file.
A library (with the .lib extension) and stk file (with the .stk extension) are generated from the specified name.

■<COMPILE OPTION>
Specify the cc32R option(s).

■<ASSEMBLE OPTION>
Specify the as32R option(s).

■<MEMORY MODEL>
Specify the memory model name (one of "small", "medium" and "large").

[Example of specification]

If library name m32RcR.lib, small model and the library corresponds to time critical optimization is generated.

```
set TARGET=m32RcR
set CFLAGS=-small -Otime -zdiv
set AFLAGS=
set MMODEL=small
```

In the EWS version, quote the detail of specification with single quote marks, etc, and specify as follows:

```
set CFLAGS='-small -Otime -zdiv'
```

<Reference>

List of option specifications corresponding to items in library supplied as standard.

Library name	**TARGET**	**CFLAGS**	**AFLAGS**	**MMODEL**
m32RcR.lib	m32RcR	-small -Otime -zdiv	None	small
m32RcRm.lib	m32RcRM	-medium -Otime -zdiv	None	medum
m32cRL.lib	m32RcRL	-large -Otime -zdiv	None	large

(2) Execute BUILD

Move the current directory to lib32R\src and execute BUILD.bat from the command line.

After execution has completed, the following two files are generated if set TARGET=m32RcR is specified in (1).

m32RcR.lib (Library file)

m32RcR.stk (stk file of all of m32RcR.lib)

(3) Execute CLEAN (if necessary)

If necessary, execute CLEAN.bat.

Deletes all *.mo files temporarily created when executing BUILD, as well as individual *.stk files.

(The library and stk file for the whole library are not deleted.)

9.3 Library Function Descriptions

This section describes the functions of the C standard library in alphabetical order. How to refer each function description is shown in Figure 9.2 :

Function name

Group[Note)]

Summary

Syntax
Shows a call type of function. `#include` <*header_file*> is a standard header file to use this function. Be sure to include it.

Return Value
Shows a return value of function. A comment preceded by ' : ' which is stated immediately after a return value explains the return value (return conditions, etc.).

Description
Explains the specifications of the function.

Caution
Shows something to be considered, if any.

Figure 9.2 C Standard Library Function Reference Format

Note) The parenthesized "non-ANSI" here means a function that is included in the C standard library and it however is not defined in ANSI-C. (see Sections 7.3.6, 7.3.7 and 7.3.8).

_action_atexit

Termination function (non-ANSI)

Performs user-registered terminations.

Syntax

```
void _action_atexit ( void );
```

Return Value

None

Description

The _action_atexit function calls all of termination functions registered by using the atexit function, and then, clears the counter which counts registered functions.

Caution

Call this function always after terminating the main function in the start-up program. (See 7.2, 7.3.)

_exit_mem

Termination function (non-ANSI)

Terminating for I/O, general utility, and localization groups.

Syntax

```
void _exit_mem ( void );
```

Return Value

None

Description

The _exit_mem function deallocates the whole memory space allocated, and initializes the memory management pointer. Call this function only once after all of input/output, general utility, and localization functions terminate.

Caution

Call this function always after terminating the main function in the start-up program. (See 7.2, 7.3.)

_exit_stdio

Termination function (non-ANSI)

Terminating for I/O function group.

Syntax

```
void _exit_stdio ( void );
```

Return Value

None

Description

The _exit_stdio function closes all of input/output streams. Call this function only once after all of input/output functions terminate.

Caution

Call this function always after terminating the main function in the start-up program. (See 7.2, 7.3.)

_get_exit_code

Termination function (non-ANSI)

Gets the exit status from exit().

Syntax

```
void _get_exit_code ( void );
```

Return Value

The exit status (the termination code) returned from the exit function.

Description

The _get_exit_code function gets the exit status code from the exit function. Call this function always in the start-up program to check the exit status for the function which has been terminated with the exit function. (See 7.3.9 "Start-up Program Example.")

_init_atexit

Initialization function (non-ANSI)

Initializing for atexit().

Syntax

```
void _init_atexit ( void );
```

Return Value

None

Description

The _init_atexit function initializes the counter which counts registered functions by the _atexit function. Call this function only once before the first atexit function call.

Caution

Call this function always before calling the main function in the start-up program. See _action_atexit(). (See 7.2, 7.3.)

_init_base_year

Initialization function (non-ANSI)

Initializing for the date and time function group.

Syntax

```
void  _init_base_year ( int year );
     year;          /* base year */
```

Return Value

None

Description

The _init_base_year function initializes the base year to *year*. Call this function only once prior to all of the functions of the date and time function group.

Caution

Call this function always before calling the main function in the start-up program. (See 7.2, 7.3.)

_init_exit_environ

Initialization function (non-ANSI)

Initializing for exit().

Syntax

```
void _init_exit_environ ( void );
```

Return Value

0 : The status as it is before the function is called (the current environment has been saved).

non-0 : Control has returned as a result of executing the exit function (this return value turns to 1 if the exit status code of the routine that issued this function is 0).

Description

The _init_exit_environ function saves the current execution environment by using the setjmp function. This initializes the return destination handled by the exit function. Having called this function before executing the main function within the start-up program allows control to return to the point at the time of issuing this function when the main function terminates with the exit function.

This function returns a value other than 0 if returned from the exit function, otherwise returns 0.

In the start-up program, calling this function before calling the main function and checking its return value enables the following (see also 7.3.9 "Start-up Program Examples") :

- Return Value = 0 This means that the initialization of the return destination from the exit function has been finished, so you can call the main function.
- Return Value = a value other than 0
  This means that the main function has terminated with the exit function and that control has returned to the point at the time of issuing this function. you can obtain the exit status of the exit function through the _get_exit_code function.

_init_mem

Initialization function (non-ANSI)

Initializing for I/O, general utility, and localization groups.

Syntax

```
void _init_mem ( void );
```

Return Value

None

Description

The _init_mem function initializes the memory management pointer. Call this function only once prior to all of input/output, general utility, and localization functions.

Caution

Call this function always before calling the main function in the start-up program. The corresponding termination process with the _exit_mem function is required after control returns from the main function. (See 7.2, 7.3.)

_init_stdio

Initialization function (non-ANSI)

Initializing for I/O function group.

Syntax

```
void _init_stdio ( void );
```

Return Value

None

Description

The _init_stdio function effect the initialization to use the standard input stream, standard output stream, and standard error stream. Call this function only once prior to all of the functions of the input/output function group.

Caution

Call this function always before calling the main function in the start-up program. The corresponding termination process with the _exit_stdio function is required after control returns from the main function. (See 7.2, 7.3.)

abort

General utility function

Puts the running program to forced stop.

Syntax

```
#include  < stdlib.h >
void abort  ( void );
```

Return Value

None

Description

The abort function terminates execution of the program and generates a signal SIGABRT. Abort function does not flush any buffer, close any file nor delete any temporary file.

abs

General utility function

Obtains the absolute value of an int type integer.

Syntax

```
#include  < stdlib.h >
int  abs ( int j );
     j;                 /* an integer to be computed */
```

Return Value

An absolute value of the result

Description

The abs function calculates an absolute value of int type. Any operation cannot be guaranteed if the result cannot be expressed in the int type.

acos

Mathematics function

Obtains the arc cosine of a floating-point number.

Syntax

```
#include  < math.h >
double  acos ( double x );
     x;
```

Return Value

Calculated value of argument

Description

The acos function calculates the arc cosine of argument *x* and returns a value in a range from 0 to π. Value *x* must be in a range between -1 and 1. This function sets a value of EDOM in errno if *x* is less than -1 or more than 1.

asctime

Date and time function

Converts data and time (a struct tm) into the equivalent text string.

Syntax

```
#include < time.h >
char *asctime ( const struct tm *timeptr );
     timeptr;       /* pointer to tm type structure */
```

Return Value

Pointer for a converted character string

Description

The asctime function converts the time and date specified by *timeptr* into the format as follows :

Thu May 12 16:00:00 1995\n\0

asin

Mathematics function

Obtains the arc sine of a floating-point number.

Syntax

```
#include  < math.h >
double asin ( double x );
     x;
```

Return Value

Calculated value of argument

Description

The asin function calculates the arc sine of argument *x* and returns a value in a range from -π/2 to π/2. Value *x* must be in a range between -1 and 1. This function sets a value of EDOM in errno if *x* is less than -1 or more than 1.

assert

Program diagnostic function

Adds a diagnostic function to a program.

Syntax

```
#include  < assert.h >
#include  < stdio.h >
void  assert ( int expression );
     expression;         /* an expression to be diagnosed */
```

Return Value

None

Description

The assert function is defined as a macro and it outputs a diagnosis message to a standard error file when *expression* is false (0). And then the assert function starts the abort function. This function does nothing when *expression* is true.

The assert function is usually used to detect a logical error of the program and a conditional *expression* is specified so that expression can be true only when a program runs correctly.

If the description of assert is deleted from the program after the program has been debugged, define a macro called NDEBUG in the #define statement before fetching <assert.h> (#define NDEBUG).

If the #undef statement is used for a macro called assert, the effect of the assert function is not guaranteed any more.

atan

Mathematics function

Obtains the arc tangent of a floating-point number.

Syntax

```
#include  < math.h >
double  atan ( double x );
     x;
```

Return Value

Calculated value of argument

Description

The atan function calculates the arc tangent of *x*. This function returns a value in a range from 0 to π.

atan2

Mathematics function

Divides a floating-point number by a floating-point number and obtains the arc tangent of the result.

Syntax

```
#include  < math.h >
double  atan2 ( double y,  double x );
     y, x;
```

Return Value

Calculated value of argument

Description

The atan2 function calculates the arc tangent of *y*/*x*. This function returns a value in a range from -π to π. The result of the atan2 function indicates an angle made by the *x* axis of a memory area through a point (*x*, *y*) and origin (0, 0). When both arguments, *x* and *y*, of the atan2 function is 0, this function sets a value EDOM in errno.

atexit

General utility function

Catalogs the function to be called upon successful termination of the program.

Syntax

```
#include < stdlib.h >
int atexit ( void (*func)(void) );
     func;     /* pointer to registered function */
```

Return Value

0	: Registered
Other than 0	: Not registered

Description

The atexit function registers a function called when a program has completed correctly or when the exit function has been called. A maximum of 32 functions can be registered. When the program has completed, the functions are executed in the reverse order of registration.

atof

General utility function

Converts the character string representing a number into a double type floating-point number.

Syntax

```
#include  < stdlib.h >
double  atof ( const char *nptr );
     nptr;          /* pointer to string to be converted */
```

Return Value

Converted value	: Successful

Description

The atof function converts a character string nptr expressing a number into a double type value and returns a converted value. If a conversion result overflows or underflows, this function sets a value of ERANGE in errno and returns HUGE_VAL (-HUGE_VAL if negative) or 0 as a return value.

atoi

General utility function

Converts the character string representing a decimal number into a int type integer.

Syntax

```
#include  < stdlib.h >
int  atoi ( const char *nptr );
     nptr;          /* pointer to string to be converted */
```

Return Value

Converted value : Successful

Description

The atoi function converts a character string *nptr* expressing a number into an int type value and returns a converted value. If a conversion result overflows, this function sets a value of ERANGE in errno and returns INT_MAX (INT_MIN if negative) or 0 as a return value.

atol

General utility function

Converts the character string representing a decimal number into a long type integer.

Syntax

```
#include  < stdlib.h >
long  atol ( const char *nptr );
     nptr;          /* pointer to string to be converted */
```

Return Value

Converted value : Successful

Description

The atol function converts a character string *nptr* expressing a number into an long type value and returns a converted value. If a conversion result overflows, this function sets a value of ERANGE in errno and returns LONG_MAX (LONG_MIN if negative) or 0 as a return value.

bsearch

General utility function

Performs binary search.

Syntax

```
#include  < stdlib.h >
void  *bsearch ( const void *key, const void *base,
                 size_t nmemb, size_t size,
                 int (*compar )(const void *, const void *) );

     key;      /* pointer to data to be surched */
     base;     /* pointer to table to be surched */
     nmemb;    /* the number of members to be surched */
     size;     /* bytes of a member */
     compar;   /* pointer to function to be compared */
```

Return Value

Pointer for specified members	: Specified members have been searched
NULL	: Not searched

Description

The bsearch function searches for members who meet the conditions of data specified by *key* through a table specified by *base*. A function which performs comparison receives pointers p1 (the 1st argument) and p2 (the 2nd argument) for the two comparison data and returns the result according to the following specifications :

- *p1 < *p2 this function returns a negative value.
- *p1 == *p2 this function returns a value.
- *p1 > *p2 this function returns a positive value.

All the members to be searched must be arranged in ascending order.

calloc

General utility function

Allocates a memory space and initializes the allocated memory space to 0.

Syntax

```
#include  < stdlib.h >
void  *calloc ( size_t nelem, size_t elsize );
     nelem;    /* the number of elements */
     elsize;   /* bytes of an element */
```

Return Value

Top element address in allocated memory area	:	Successful
NULL	:	Error (No memory area has been allocated. Any of the arguments is 0.)

Description

The calloc function allocates *nelem* objects of the memory area in *elsize* bytes . This function initializes all of the allocated area to 0.

Note

12 bytes more area have been secured as memory area allocated by each the calloc functions. This memory area of 12 bytes is stored a allocation infomation (size etc.).

ceil

Mathematics function

Computes the integer ceiling of a floating-point number.

Syntax

```
#include  < math.h >
double  ceil ( double x );
     x;        /* floating-point number */
```

Return Value

The calculation result

Description

The ceil function returns a minimum integer equal to or greater than value *x* as double type value.

clearerr

Input/output function

Clears an error condition in a stream.

Syntax

```
#include  < stdio.h >
void  clearerr ( FILE *fp );
     fp;       /* pointer to FILE structure */
```

Return Value

None

Description

The clearerr function clears the error and EOF (end-of-file) indicators for the stream pointed to by *fp*. The indicators are data held per stream file. Both data indicate whether an error is occurred or not and whether a file is completed or not and such data can be referred through the ferror and feof function respectively. If any information on the error occurrence and file completion cannot be obtained through the return value of the functions which handle the stream, this function checks the status of the file by means of these data.

clock

Date and time function

Gets the elapsed processor time.

Syntax

```
#include  < time.h >
clock_t  clock ( void );
```

Return Value

Not supported

Description

The clock function depends on your system environment. Therefore, the clock function is not supported for the present.

Caution

To call the clock function in actual, the user-written clock function is required (see Section 11.1 and Table 11.2).

cos

Mathematics function

Obtains the cosine of radians of a floating-point number.

Syntax

```
#include  < math.h >
double  cos ( double x );
     x;        /* floating-point number (in radians) */
```

Return Value

Calculated value of argument

Description

The cos function calculates cosine of x.

cosh

Mathematics function

Obtains the hyperbolic cosine of a floating-point number.

Syntax

```
#include  < math.h >
double  cosh ( double x );
     x;        /* floating-point number */
```

Return Value

Calculated value of argument : Successful

Description

The cosh function calculates the hyperbolic cosine of *x*. When the calculation result of the function cannot be expressed as a double type value, this function sets a value of ERANGE in errno. If the calculation result overflows, this function returns HUGE_VAL as a return value.

ctime

Date and time function

Converts the calendar time (a time_t value) into the equivalent text string.

Syntax

```
#include  < time.h >
char  *ctime ( const time_t *timer );
     timer;         /* calendar time */
```

Return Value

Pointer to a converted character string

Description

The ctime function converts the calendar time specified by *timer* into the following format :

Thu May 12 16:00:00 1995\n\0

difftime

Date and time function

Computes the difference between the two specified times.

Syntax

```
#include  < time.h >
double  difftime ( time_t time1, time_t time2 );
     time1;         /* time 1 */
     time2;         /* time 2 */
```

Return Value

The value of *time2* subtracted from *time1* (by the second)

Description

The difftime function calculates the value of *time2* subtracted from *time1*.

div

General utility function

Divides an int type integer and obtains the quotient and remainder.

Syntax

```
#include  < stdlib.h >
div_t  div ( int number, int denom );
     number;        /* dividend */
     denom;         /* divisor */
```

Return Value

Quotient and the remainder in calculation result

Description

The div function calculates the quotient and the remainder for the result of dividing *number* by *denom*. The return value is `div_t` structure and the quotient and the remainder are put in `quot` and `rem` respectively of the member.

exit

General utility function

Terminates the program.

Syntax

```
#include  < stdlib.h >
void  exit ( int status );
     status;        /* exit status code */
```

Return Value

None

Description

The exit function terminates a program by executing the following :

(1) Flashes all the buffers of an opened stream.
(2) Closes all the opened files.
(3) Deletes all the files generated by tmpfile.
(4) Executes all the functions registered by atexit functions in reverse order.
(5) Returns control to the environment saved by a setjmp function.
(Control no returns to the caller.)

To check *status* in the environment to which the process returns, you can use the _get_exit_code function.

exp

Mathematics function

Obtains the exponential function of a floating-point number.

Syntax

```
#include  < math.h >
double  exp ( double x );
     x;        /* floating-point number */
```

Return Value

Value of exponential function (e^x) : Successful

Description

The exp function calculates the exponential function (e^x) of *x* and returns the result. If a conversion result overflows, this function sets a value of ERANGE in errno and returns HUGE_VAL as a return value.

fabs

Mathematics function

Obtains the absolute value of a floating-point number.

Syntax

```
#include  < math.h >
double  fabs ( double x );
     x;        /* floating-point number */
```

Return Value

Absolute value of *x*

Description

The fabs function returns the absolute value of *x*.

fclose

Input/output function

Closes a file.

Syntax

```
#include  < stdio.h >
int  fclose ( FILE *fp );
     fp;       /* pointer to FILE structure */
```

Return Value

0	: Successful
EOF	: Error

Description

The fclose function closes the I/O stream specified by file pointer *fp*. If an output file of the I/O stream is opened and if any data not yet output still remains in the buffer, this function outputs it to a file and closes the file. If the I/O buffer is assigned automatically by a system, this function release it. Moreover, if the stream is opened by the tmpfile function, this function deletes the corresponding file.

feof

Input/output function

Checks if the end of a stream is reached.

Syntax

```
#include  < stdio.h >
int  feof ( FILE *fp );
     fp;       /* pointer to FILE structure */
```

Return Value

0	: When a file is not ended
non-0	: When a file is ended

Description

The feof function judges whether the I/O stream specified by the file pointer *fp* is ended or not. The feof function judges that the file is ended if the EOF (end-of-file) indicator is set.

ferror

Input/output function

Checks if a stream is in an error condition.

Syntax

```
#include  < stdio.h >
int  ferror ( FILE *fp );
     fp;       /* pointer to FILE structure */
```

Return Value

0	: No error condition for the file
non- 0	: An error condition for the file

Description

The ferror function judges whether any error is made in the I/O stream specified by the file pointer *fp*. The ferror function judges that an error is made in the file if the error indicator is set.

fflush

Input/output function

Outputs the contents of a stream to a file.

Syntax

```
#include  < stdio.h >
int  fflush ( FILE *fp );
     fp;       /* pointer to FILE structure */
```

Return Value

0	: Successful
EOF	: Error

Description

When the stream specified by *fp* is opened for output, the fflush function outputs the stream buffer content, which has not yet been output, to the file. This function disables the designation of the ungetc function if such a file is opened for input.

fgetc

Input/output function

Gets a character from a stream.

Syntax

```
#include  < stdio.h >
int  fgetc ( FILE *fp );
     fp;       /* pointer to FILE structure */
```

Return Value

The character read	: Successful
EOF	: End-of-file is encountered or a read error occurs.

Description

The fgetc function inputs one character from the stream specified by the file pointer *fp* and returns the character.

When EOF (end-of-file) is encountered, the EOF indicator for the stream is set and this function returns EOF. When a read error occurs during the operation, the error indicator for the stream is set and this function returns EOF.

The EOF indicator can be referred by the feof function and the error indicator can be referred by the ferror function.

fgetpos

Input/output function

Locates the current position on a stream.

Syntax

```
#include  < stdio.h >
int  fgetpos ( FILE *stream, fpos_t *pos );
     stream;        /* pointer to FILE structure */
     pos;           /* current position on the stream */
```

Return Value

0	: Successful
non- 0	: Error (An error is returned to errno.)

Description

The fgetpos function returns the current value of the file position indicator of the stream specified by *stream* to the memory area specified by *pos*.

fgets

Input/output function

Gets a string from an input stream.

Syntax

```
#include  < stdio.h >
char  *fgets ( char *s, int n, FILE *fp );
     s;        /* pointer to data storage location */
     n;        /* the number of characters to be stored */
     fp;       /* pointer to FILE structure */
```

Return Value

s	: Successful
NULL	: End-of -file is encountered or a read error occurs.

Note) The content of the storage specified by *s* does not vary when an input file is ended, but it depends on the user-defined read function when an error occurs.

Description

The fgets function inputs a string from the stream specified by the file pointer *fp* and stores it in the memory area specified by the pointer *s*. This function inputs *n*-1 characters, any characters up to the new-line character or to the end of the file and it adds a null character to the end of the string. The input new-line character is included in the string and is stored.

When EOF (end-of-file) is encountered, the EOF indicator for the stream is set and this function returns EOF. When a read error occurs during the operation, the error indicator for the stream is set and this function returns EOF.

The EOF indicator can be referred by the feof function and the error indicator can be referred by the ferror function.

floor

Mathematics function

Cuts off the fraction of a floating-point number.

Syntax

```
#include  < math.h >
double  floor ( double x );
     x;        /* floating-point number */
```

Return Value

Calculation result

Description

The floor function returns the maximum integer equal to or less than the value *x* as a double type value.

fmod

Mathematics function

Computes the floating-point remainder.

Syntax

```
#include  < math.h >
double  fmod ( double x, double y );
     x, y;          /* floating-point number */
```

Return Value

The remainder of *x*/*y*

Description

The fmod function calculates the remainder of *x*/*y* and returns the result as a double type. The return value (assumed as *f*) has the same sign as the dividend *x* , the absolute value of *f* is smaller than the absolute value of the divisor *y*, and the following expression is satisfied :

$x=y*i+f$ where, *i* is some integer.

If the quotient of *x*/*y* cannot be expressed (e.g., when *y*=0), any result is not guaranteed.

fopen

Input/output function

Opens a file.

Syntax

```
#include  < stdio.h >
FILE  *fopen ( const char *fname, const char *mode );
     fname;         /* file name */
     mode;          /* file access mode */
```

Return Value

Pointer to opened stream : Successful
NULL : Error

Description

The fopen function opens a file specified by *fname*. Set any of the following file access modes to the argument *mode* :

Access Mode	Meaning
"r"	Opens a file for reading
"w"	Opens a file for writing
"a"	Opens a file for appending
"r+"	Opens a file for reading and updating
"w+"	Opens a file for writing and updating
"a+"	Opens a file for appending and updating

A file is opened in a text mode. The file can be opened in a binary mode by adding b after the access mode designation (Example : r+b).

If any file to be opened is not found in the writing mode or reading mode, a new file is created.

If any existing file is opened in the writing mode, the beginning of the file is overwritten and the former record is erased.

If any file is opened in the appending mode, writing is performed from the end of the file.

If any file is opened in the updating mode, I/O processing is available for the file. However, input processing cannot be continued without executing the fflush, fseek or rewind function after output processing. Moreover, the output processing cannot be continued without executing these functions after the input processing.

fprintf

Input/output function

Outputs data to a stream according to the format.

Syntax

```
#include  < stdio.h >
int  fprintf ( FILE *fp, const char *control, ... );
     fp;       /* pointer to FILE structure */
     control;  /* pointer to format string */
     ...;      /* variable argument list (output data) */
```

Return Value

The number of output characters	: Successful
Negative value	: Error

Description

The fprintf function converts and edits the output data specified by the variable argument list according to the format *control* and then outputs them to the I/O stream specified by the pointer *fp* .

Details of the format are (Ditto for formats handled in the printf, sprintf, vfprintf, vprintf and vsprintf functions.) :

<Contents of the Format>

A format is character string (pointed to by *control*) which can contain two kinds of character sequences such as “plain characters” and “conversion specification”. The number of and the appearance order of conversion specifications correspond to them of arguments in the variable argument list.

- **Plain characters**

  Characters in a sequence beginning with non-% (i.e., other than a conversion specification) . They are output directly.

  Example : `fprintf(fp,"data=%02d", a);`
  The underlined are plain characters.

- **Conversion specification**

  A character sequence beginning with %. It specifies how to convert the corresponding argument in the variable argument list. It consists of % and following specifiers shown below (They are optional.) :

- Flags
- Field width
- Precision
- Size specifier (for the corresponding argument)
- Conversion specifier

Example : `fprintf(fp,"data=%02d", a);`
The underlined is a conversion specification. It converts `a`.

<Conversion Specification Syntax>

A conversion specification can be specified by the following format (a specifier enclosed in [ ] is optional) :

`%` [*flags*] [*field_width*] [`.` [*precision*]] [*size_specifier*] *conversion_specifier*

Example : `%02d`
The %, the flag 0, the field width 2, and the conversion specifier `d`.

Every item shall be described continuously (not separated by a space). If there is no corresponding argument for a conversion specification (such as : there is no variable argument list. arguments are not enough.), the operation is not guaranteed. If the number of arguments is larger than the number of conversion specifications, all of the excessive arguments are ignored.

<Details of Conversion Specifiers>

The function of each conversion specifier and how to specify are :

- **Flag**

A flag specifies processing for the data to be output, such as marking with a symbol. Types and meaning of the available flags are described as follows :

<table>
<tr><th>Flag</th><th>Meaning</th></tr>
<tr><td><code>'-'</code></td><td>Left-justified. If the number of characters of converted data is smaller than the specified field width, this function outputs the data as left-justification in the filed. If this flag is not specified, this function will output the data as right-justification.</td></tr>
<tr><td><code>'+'</code></td><td>With a sign. Upon converting into data followed by a sign, this function adds the sign '+' or '-' to the beginning of the converted data.</td></tr>
<tr><td><code>' '</code> (space)</td><td>Without a sign. Upon converting into data followed by a sign, a space is added to the beginning of converted data if no sign is added to the beginning of the data. This flag is ignored if it is specified together with '+'.</td></tr>
<tr><td><code>'#'</code></td><td>Processing. This function processes the converted data according to the conversion specifier as follows :
<table>
<tr><th>Conversion Specifier</th><th>Processing</th></tr>
<tr><td><code>c,d,i,s,u</code></td><td>This flag is ignored.</td></tr>
<tr><td><code>o</code></td><td>0 is added to the beginning of converted data if the data is not 0.</td></tr>
<tr><td><code>x,X</code></td><td>0x (or 0X if X conversion) is added to the beginning of the converted data if the data is not 0.</td></tr>
<tr><td><code>e,E,f,g,G</code></td><td>A floating-point is output even if the converted data has no fractional part. When g or G, 0 which is appended to the converted data is not removed.</td></tr>
</table></td></tr>
<tr><td>0</td><td>If a conversion specifier is d, i, o, u, x, X, e, E, f, g or G, a space at the left of value is filled with several 0s to eliminate the space. If it is specified together with the flag '-' or if a conversion specifier is d, i, o, u, x or X, this flag is ignored when any precision is specified.</td></tr>
</table>

- **Field width**

  A field width specifies the number of output characters of converted data in decimal number format or `*` (asterisk).

  If the number of output characters of the converted data is smaller than the field width, several spaces added to the beginning of the data to adapt to the field width. However, if the flag '-' is specified, the spaces are appended to the data.

  If the number of output characters of converted data is larger than the field width, the field width is expanded so that the conversion result can be output.

  If the flag '0' is specified, a character '0' instead of space is added to the beginning of the output data.

- **Precision**

  A precision specifies the precision of the converted data according to the kind of conversion specifier.

  Upon specifying, write the decimal integers and the * after a period (.).

  If the decimal integers are omitted, it is assumed that 0 has been specified. As the result of the accuracy specification, if there is any discrepancy between the field width and the specification, the field width specification is disabled.
  The type of conversion specifier and the meaning of the precision specification are stated below :

Conversion specifier	Precision specification
`d,i,o,u,x,X`	Shows the minimum number of digits of converted data.
`e,E,f`	Shows the number of digits of fractional part of converted data.
`g,G`	Shows the maximum number of effective digits of converted data.
`s`	Show the maximum characters to be printed.

- **The * specification for field width or precision**

  An `*` (asterisk) can be used to specify the field width or precision.

  When `*` is specified, the value of the variable argument corresponding to the conversion specification is used as a value to specify the field width or precision. If the value of the variable argument is negative, the field width `*` is assumed that the '`-`' flag has been specified for a positive field width. The precision `*` is assumed that the precision has been omitted.

- **Size specifier (for the corresponding argument)**

With the size specifiers h, l, or L at the preceding a conversion specifier, size of the corresponding argument can be specified when the conversion specifier is d, i, o, u, x, X, e, E, f, g, G or n. Types of the size specifiers and their meanings are shown below:

Size specifier	Following conversion specifier	Type of the corresponding argument is...
h	d,i,o,u,x,or X	short int, or unsigned short int
	n	pointer to short int
l(ell)	d,i,o,u,x,or X	long int, or unsigned long int
	n	pointer to long int
L	e,E,f,g,or G	long double

A size specifier is ignored, if the following conversion specifier is other than d, i, o, u, x, X, e, E, f, g, G or n.

- **Conversion specifier**

A conversion specifier specifies how the corresponding argument is converted.

If the argument to be converted is a structure or array type or any pointer for those types, the operation is not guaranteed (except upon converting character array by s conversion and upon converting pointer by p conversion).

The conversion specifiers and conversion methods are shown below (Tables). If specifying any character not mentioned here as the conversion specifier, the operation is not guaranteed.

Notice in the tables : Types in the “Type of the Argument” columns are shown, assuming that a size specifier (h, l, or L) is not specified in the conversion specification. “The argument” and “the precision” mean an argument (in the variable argument list) and the precision in the conversion specification, respectively, correspond to the conversion specifier.

(Conversion Specifiers for `fprintf` (1/3))

Conversion Specifier	Conversion Method	Type of the Argument	Notes on Precision
d or i	Converts int type data into a character sequence of signed decimal notation. The conversion specifier d behaves the same as i.	int	The precision shows how many characters are output at least. If the number of converted characters is smaller than the value of precision, 0 is added to the beginning of the character string. If the precision is omitted, 1 is assumed. Even if any data of value 0 is output with a precision of 0 and converting it, nothing is output.
o	Converts unsigned int type data into a character sequence of unsigned octal notation.	unsigned int	
u	Converts unsigned int type data into a character sequence of unsigned decimal notation.	unsigned int	
x	Converts unsigned int type data into a character sequence of unsigned hexadecimal notation. The hexadecimal letters `abcdef` are used.	unsigned int	
X	Converts unsigned int type data into a character sequence of unsigned hexadecimal notation. The hexadecimal letters `ABCDEF` are used.	unsigned int	
f	Converts double type data into a character sequence of decimal notation in the format [-]*ddd*.*ddd*.	double	The precision shows the number of digit of fractional part. If there is (are) any decimal place(s), a figure is output before a decimal point. If the precision is omitted, 6 is assumed. If the precision is 0, a fractional part is not output. Output data is rounded.

(Conversion Specifiers for `fprintf` (2/3))

Conversion Specifier	Conversion Method	Type of the Argument	Notes on Precision
e	Converts double type data into a character sequence of decimal notation in the format [-]*d*.*ddd*e ± *ddd*. The exponent is output at least two digits.	double	The precision shows the number of digit of fractional part. For a converted character, one figure is output before a decimal point and the number of digits equal to the precision is output after the decimal point. If the precision is omitted, 6 is assumed. If the precision is 0, a decimal point and decimal place are not output. Output data is rounded.
E	Converts double type data into a character sequence of decimal notation in the format [-]*d*.*ddd*E ±*ddd*. The exponent is output at least two digits.	double	
g (or G)	Outputs the value (double type) stored int the argument in either the f conversion format or in the e (or E) conversion format depending on a value stored in the argument and the precision which specifies the number of effective digits. If the exponent of converted data is smaller than -4 or more than the precision, this function converts to the e (or E) conversion format. After the conversion, a 0 at the end of the decimal places is removed. If there is no character at any of the decimal places, the decimal point is also removed.	double	The precision shows the maximum number of effective digits of converted data.
c	Considers int type data as unsigned char and converts it into a character corresponding to the data.	int	The precision is disabled.

(Conversion Specifiers for `fprintf` (3/3))

Conversion Specifier	Conversion Method	Type of the Argument	Notes on Precision
s	Outputs a character sequence specified by data of the pointer to char type up to the null character which shows the end of the character string or the output the number of characters specified by the precision. (The null character is not output.)	pointer to char	The precision shows the number of characters to be output. If the precision is omitted, the character string specified by the data up to the null character is output. (The null character is not output.)
p	Regards data as pointer type and converts it into a printable character string depending on the compiler.	pointer to void	The precision is disabled.
n	Considers the data is regarded as pointer type to int type and sets the number of characters of data output to the memory area specified by the data.	pointer to int	The precision is disabled.
%	Outputs %.	None.	The precision is disabled.

Note) Specifying the n or % specifier cannot convert the corresponding argument.

fputc

Input/output function

Outputs a character to a stream.

Syntax

```
#include  < stdio.h >
int  fputc ( int c, FILE *fp );
     c;        /* character(s) to be output */
     fp;       /* pointer to File structure */
```

Return Value

The character written	: Successful
EOF	: A write error occurs.

description

The fputc function outputs the character *c* into the stream specified by the file pointer *fp*.

When a write error occurs during the operation, the error indicator for the stream is set and this function returns EOF.

The error indicator can be referred by the ferror function.

fputs

Input/output function

Outputs a string to a stream.

Syntax

```
#include  < stdio.h >
int  fputs ( char *s, FILE *fp );
     s;        /* pointer to character(s) to be output */
     fp;       /* pointer to File structure */
```

Return Value

0	: Successful
EOF	: A write error occurs.

Description

The fputs function outputs the character(s) specified by s into the stream specified by the file pointer *fp*. However, the null character is not output.

When a write error occurs during the operation, the error indicator for the stream is set and this function returns EOF.

The error indicator can be referred by the ferror function.

fread

Input/output function

Transfers data from a stream to a memory area.

Syntax

```
#include  < stdio.h >
size_t  fread ( void *ptr, size_t size, size_t n, FILE *fp );
     ptr;      /* pointer to data storage area */
     size;     /* the number of bytes of one member */
     n;        /* the number of members to be read */
     fp;       /* pointer to File structure */
```

Return Value

The number of read members (the same value as *n* usually)	: Successful
A value smaller than *n*	: The file is ended or an error occurs (It can be judged by the ferror and feof function)

Description

The fread function reads up to *n* members which have *size*-byte data from the stream pointed to by *fp* to the memory area specified by *ptr*.

If *size* or *n* is 0, this function returns 0 as a return value and the contents of the memory area specified by *ptr* does not change. If an error in encountered or if the members are not read completely, the value of the file position indicator is not guaranteed.

free

General utility function

Releases the specified memory area.

Syntax

```
#include  < stdlib.h >
void  free ( void *ptr );
     ptr;      /* top address of memory area to be free */
```

Return Value

None

Description

The free function frees the memory area specified by *ptr* so that the memory area can be allocated again for use. This function does nothing if *ptr* is NULL.

If the memory area to be free by the free function or the memory area specified by *ptr* of the realloc function is not any of memory area allocated by the calloc, malloc and realloc functions, or if the memory area has been already freed by the free and realloc functions, the operation is not guaranteed.

freopen

Input/output function

Closes a currently opened stream, and reopens a new file with the new file name.

Syntax

```
#include  < stdio.h >
FILE  *freopen ( const char *fname, const char * mode,
FILE *fp );
     fname;         /* file name */
     mode;          /* file access mode */
     fp;            /* pointer to File structure */
```

Return Value

Pointer for opened stream	: Successful
NULL pointer	: Error

Description

The freopen function closes the stream pointed to by *fp* opened currently and opens a new file *fname* for the stream by reusing the FILE structure specified by the same *fp*. Select any file access mode from the access modes used in the fopen function and set it as the mode.

frexp

Mathematics function

Divides a floating-point number into products of value (0.5, 1.0) and 2 to the *n*th power.

Syntax

```
#include  < math.h >
double  frexp ( double x, int *e );
     x;   /* floating-point number */
     e;   /* pointer to area in which exponent (integer) is */
```

Return Value

Value of coefficient *m*

Description

The frexp function calculates a power (*n*) of 2, which meets the condition, $x=m*2^n$, and the coefficient (*m*), where the absolute value of the coefficient *m* is smaller than 1.0 and larger than 0.5. A power value *n* is stored in the memory area specified by the argument *e* and the coefficient *m* is returned as a return value. If x is 0.0, both *m* and *n* are 0.0.

fscanf

Input/output function

Gets data from a stream, and converts the data by following the format.

Syntax

```
#include  < stdio.h >
int  fscanf ( FILE *fp,  const char *control , ... );
     fp;        /* pointer to File structure */
     control;  /* pointer to format string */
     ...;       /* variable argument list (receivers) */
```

Return Value

The number of arguments converted and substituted	: Successful (Substitution suppression by %n conversion and * is not included in this number)
EOF	: Input data has been completed before the first conversion (excluding %%) has been succeeded. Otherwise an error occurred.

Description

The fscanf function reads input data from the I/O stream pointed to by the file pointer *fp*, converts and edits the input according to the format *control*, and then stores the result into the memory area(s) specified by the variable argument list.

An input item (text in a file when the fscanf function) is separated into tokens by on or more white-spaces : space(' '), horizontal tab ('\t'), newline ('\n'). Those white-spaces are ignored. For example, the input " `ABCD abcd`" is translated into the 2 tokens "`ABCD`" and "`abcd`". The 2 tokens are converted and then stores into the corresponding areas specified by arguments in the variable argument list. (In this example, 2 arguments are required.)

Details of the format are shown below (Ditto for formats handled in the scanf and sscanf functions) :

<Contents of the Format>

A format is character string (pointed to by *control*) which can contain two kinds of character sequences such as "plain characters" and "conversion specification". The number of and the appearance order of conversion specifications correspond to them of arguments (in the variable argument list) which are pointers to the areas in which the separated input items store.

- **Plain characters**

  Characters in a sequence beginning with non-% (i.e., other than a conversion specification) but White-spaces. They can be input if there are the characters matched up to them in the input item (text in a file when the fscanf function). If there are characters unmatched up to them, the unmatched characters are remained in the input stream.

- **Conversion specification**

  A character sequence beginning with %. It specifies how to convert the input data. It consists of % and following specifiers shown below (They are optional.) :

  - *
  - Field width
  - Size specifier (for the corresponding argument)
  - Conversion specifier

<Conversion Specification Syntax>

A conversion specification can be specified by the following format (a specifier in [ ] is optional) :

`%[*]` [*field_width*] [*size_specifier*] *conversion_specifier*

Example : `%2d%f`
Two conversion specifications :
The %, the field width 2, and the conversion specifier d.
The % and the conversion specifier f .

Every item shall be described continuously (not separated by a space). If there is no corresponding argument for a conversion specification (such as : there is no variable argument list. there are insufficient arguments.), the operation is not guaranteed. If the number of arguments is larger than the number of conversion specifications, all of the excessive arguments are ignored.

<Details of Conversion Specifiers>

The function of each conversion specifier and how to specify are :

- *** (asterisk)**

  Prefixing * prior to a conversion specifier allows reading the corresponding token from the input but suppresses assignment.

- **Field width**

  A field width specifies the number of characters in decimal number format which can be input.

- **Size specifier (for the corresponding argument)**

With the size specifiers h, l, or L at the preceding a conversion specifier, type of the corresponding argument can be specified when the conversion specifier is d, i, o, u, x, X, e, E, f, g, G or n :

Size specifier	Following conversion specifier	Type of the corresponding argument is...
h	d,i,o,u,x,or X	short int, or unsigned short int
	n	pointer to short int
l(ell)	d,i,o,u,x,or X	long int, or unsigned long int
	n	pointer to long int
L	e,E,f,g,or G	long double

A size specifier is ignored, if the following conversion specifier is other than d, i, o, u, x, X, e, E, f, g, G or n.

- **Conversion specifier**

A conversion specifier specifies how the corresponding argument is converted.

The fscanf function, unless the conversion specifier is c, [, n or %, skips the white-spaces in the input before conversion (The skipped characters are not included in the field width).

When a space is read during conversion, if it is any character unauthorized for conversion, processing is terminated without reading the character. When the specified field width is completed during conversion, processing is terminated.

The conversion specifiers and conversion methods are shown below (Tables). If specifying any character not mentioned here as the conversion specifier, the operation is not guaranteed.

Notice in the tables : Types in the “Type of the Argument” columns are shown, assuming that a size specifier (h, l, or L) is not specified in the conversion specification. “The argument” means an argument (in the variable argument list) which corresponds to the conversion specifier.

(Conversion Specifiers for `fscanf` (1/2))

Conversion Specifier	Conversion Method	Type of the Argument
d	Converts a character sequence which represents a decimal number into integer data.	pointer to int
i	Converts a character sequence beginning with a sign or followed by u (or U) or l (or L) in decimal digit into integral data. Considers a character sequence beginning with 0x (or 0X) as a hexadecimal digit and converts the character sequence into integral data. Considers a character sequence beginning with 0 as an octal digit and converts the character sequence into integral data.	pointer to int
o	Converts a character sequence which represents an octal digit into integral data.	pointer to unsigned int
u	Converts a character sequence which represents an unsigned decimal digit into integral data.	pointer to unsigned int
x or X	Converts a character sequence which represents a hexadecimal digit into integral data. The x and X conversion specifiers are the same.	pointer to unsigned int
s	Converts the initial part preceding the first appearance white-space (either a space, a horizontal tab or a newline) into a character string suffixing the null character. (The area pointed to the corresponding argument should be enough to store it including the null character.)	pointer to char
c	Inputs one character. Then, the input character is not skipped even if the character is a white-space. Specifying the field width allows to read specified number of characters. Therefore the area pointed to the corresponding argument should be enough to store the characters.	pointer to char
e, E, f, F, g or G	Converts a character sequence showing a floating-point number into the floating type data. Input format is a floating-point number which can be expressed by the strtod function.	pinter to float

(Conversion Specifiers for `fscanf` (2/2))

Conversion Specifier	Conversion Method	Type of the Argument
p	Converts a character sequence in the format to be converted by the p conversion in the fprintf function into pointer type data.	pointer to void
n	No data is input and the number of characters which have been input is set.	pointer to int
[	Specifies a set of characters between [and] called "scan set" (e.g., [abcd], [a-z], [^abcd]). The scan set defines characters to be read. If a scan set begins with non-^, the initial character sequence preceding the first appearance one unspecified in the scan set is read. If a scan set begins with ^, the initial character sequence preceding the first appearance one specified in the scan set is read. Appends the null character to the read character sequence automatically. (The area pointed to the corresponding argument should be enough to store it including the null character.)	pointer to char
%	A % is read. No conversion or assignment occurs.	None.

fseek

Input/output function

Moves the current read/write position within a stream.

Syntax

```
#include  < stdio.h >
int  fseek ( FILE *fp, long offset, int type );
     fp;       /* pointer to File structure */
     offset;   /* offset from the location specified by type*/
     type;     /* offset type */
```

Return Value

0	: Successful
non-0	: Error

Description

The fseek function moves a read-write position of the stream specified by the file pointer *fp* from a place specified by the offset type *type* to a position of offset byte. Types of offset are shown as follows :

type	**Meaning**	*offset*
SEEK_SET	Beginning of file	0, a positive value
SEEK_CUR	Current read-write position	A negative value, 0, a positive value
SEEK_END	End of file	0, a negative value

For a text file, *type* must be SEEK_SET and *offset* must be 0 or any value returned by the ftell function for the file.

The ungetc function is disabled by calling the fseek function.

fsetpos

Input/output function

Changes the current position on a stream.

Syntax

```
#include  < stdio.h >
int  fsetpos ( FILE *stream, const fpos_t *pos );
     stream;   /* pointer to File structure */
     pos;      /* a position on the stream to be modified */
```

Return Value

0	: Successful
non-0	: Error (Returns an error number to errno.)

Description

The fsetpos function moves the position on the stream specified by *stream* to a place designated by memory location specified by *pos*. *pos* specifies a value returned by the fgetpos function.

ftell

Input/output function

Locates the current read / write position in a stream .

Syntax

```
#include  < stdio.h >
long  ftell ( FILE *fp );
     fp;       /* pointer to File structure */
```

Return Value

A file read-write position

Description

The ftell function seeks a read-write position of the stream specified by the file pointer *fp* and returns it as a return value.

fwrite

Input/output function

Transfers data from a memory area to a stream.

Syntax

```
#include  < stdio.h >
size_t  fwrite ( const void *ptr, size_t size, size_t n,
                  FILE *fp );
     ptr;      /* pointer to data storage area */
     size;     /* number of bytes per member */
     n;        /* number of written members */
     fp;       /* pointer to File structure */
```

Return Value

The number of read members (the same value as *n* usually)	: Successful
A value smaller than *n*	: When an error is made

Description

The fwrite function reads *n* members which have *size*-byte data from the memory area pointed to by *ptr* to the stream pointed to by *fp*.

getc

Input/output function

Gets one character from a stream.

Syntax

```
#include  < stdio.h >
int  getc ( FILE *fp );  /* one character input from file */

     fp;       /* pointer to File structure */
```

Return Value

The character read	: Successful
EOF	: End-of-file is encountered or a read error occurs.

Description

The getc function inputs one character from the stream specified by the file pointer *fp* and returns that character.

When EOF (end-of-file) is encountered, the EOF indicator for the stream is set and this function returns EOF. When a read error occurs during the operation, the error indicator for the stream is set and this function returns EOF.

The EOF indicator can be referred by the feof function and the error indicator can be referred by the ferror function.

getchar

Input/output function

Gets a character from the standard input (stdio).

Syntax

```
#include  < stdio.h >
int  getchar ( void );
               /* one character input from standard input */
```

Return Value

The character read	: Successful
EOF	: End-of-file is encountered or a read error occurs.

Description

The getchar function is the same as `getc(stdin)` function.

When EOF (end-of-file) is encountered, the EOF indicator for the stream is set and this function returns EOF. When a read error occurs during the operation, the error indicator for the stream is set and this function returns EOF.

The EOF indicator can be referred by the feof function and the error indicator can be referred by the ferror function.

getenv

General utility function

Gets the content of an environmental variable.

Syntax

```
#include  < stdlib.h >
char  *getenv ( const char *name );
     name;          /* environment variable name */
```

Return Value

Pointer to the character string specified by variable	:	Variable specified by name has been found
NULL	:	Variable specified by name has not been found

Description

The getenv function searches for the variable specified by *name* in an environment list.

Caution

To call the getenv function in actual, the user-written getenv function is required (see Section 11.1 and Table 11.2).

gets

Input/output function

Gets a string from the standard input (stdio).

Syntax

```
#include  < stdio.h >
char  *gets ( char *s );      /* string input from
                              standard input */
     s;        /* pointer to data storage area */
```

Return Value

s	: Successful
NULL	: End-of -file is encountered or a read error occurs.

Note) The content of the storage specified by *s* does not vary when an input file is ended, but it depends on the user-defined read function when an error occurs.

Description

The gets function inputs the data of one line from the standard input file (stdin) and stores it in the memory area specified by *s*. Data of one line shall be a string from the start of the input to a new-line character (or the end of file). The new-line character in the input character is discarded and null character is appended.

When EOF (end-of-file) is encountered, the EOF indicator for the stream is set and this function returns EOF. When a read error occurs during the operation, the error indicator for the stream is set and this function returns EOF.

The EOF indicator can be referred by the feof function and the error indicator can be referred by the ferror function.

gmtime

Date and time function

Converts calendar time to Coordinated Universal Time (UTC).

Syntax

```
#include < time.h >
struct tm *gmtime ( const time_t *timer );
     timer;    /* time */
```

Return Value

Converted result (pointer to tm structure)	:	Successful
NULL	:	*timer* could not be converted into UTC

Description

The gmtime function converts the calendar time shown by *timer* into Coordinated Universal Time (UTC) and converts it to a `struct tm` (broken-down time).

isalnum

Character handling function

Judges whether a letter or decimal digit.

Syntax

```
#include  < ctype.h >
int  isalnum ( int c );
     c;        /* character to be judged */
```

Return Value

non-0	: *c* is alphanumeric (‘A’-‘Z’,‘a’-‘z’,‘0’-¨9’)
0	: *c* is anything other than alphanumeric

Description

Returns any value other than 0 if c is alphanumeric and returns 0 if c is not alphanumeric. Alphanumeric shall be defined as follows :

- Upper case 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
- Lower case 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
- Decimal digits   0 1 2 3 4 5 6 7 8 9

Caution

If a value of *c* is not included in a range expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

isalpha

Character handling function

Judges whether a letter or not.

Syntax

```
#include  < ctype.h >
int  isalpha ( int c );
     c;        /* character to be judged */
```

Return Value

non-0	: *c* is alphabetic (‘A’-‘Z’,‘a’-‘z’)
0	: *c* is non alphabetic (‘A’-‘Z’,‘a’-‘z’)

Description

Returns any value other than 0 if *c* is alphabetic and returns 0 if *c* is not alphabetic. The alphabetic shall be defined as follows :

• Upper case 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
• Lower case 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

Caution

If a value of *c* is not included in a range to be expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

iscntrl

Character handling function

Judges whether a control character or not.

Syntax

```
#include  < ctype.h >
int  iscntrl ( int c );
     c;        /* character to be judged */
```

Return Value

non-0	: *c* is control character
0	: *c* is anything other than control character

Description

Returns any value other than 0 if *c* is a control character and returns 0 if *c* is not a control character. A control character is any character other than a printable character. A printable character is a character to be indicated on a display and corresponds to the ASCII codes from 0x20 to 0x7E.

Caution

If a value of *c* is not included in a range to be expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

isdigit

Character handling function

Judges whether a decimal digit or not.

Syntax

```
#include  < ctype.h >
int  isdigit ( int c );
     c;        /* character to be judged */
```

Return Value

non-0 : *c* is decimal digit

0 : *c* is anything other than decimal digit

Description

Returns any value other than 0 if c is a decimal digit and returns 0 if c is not a decimal digit

- Decimal digits : 0 1 2 3 4 5 6 7 8 9

Caution

If a value of *c* is not included in a range to be expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

isgraph

Character handling function

Judges whether a printable character other than space.

Syntax

```
#include  < ctype.h >
int  isgraph ( int c );
     c;        /* character to be judged */
```

Return Value

non-0	: *c* is a printable character excluding space
0	: *c* is not a printable character excluding space

Description

Returns any value other than 0 if *c* is a printable character excluding space (' ') and returns 0 if *c* is not a printable character or is a space. "Printable character excluding space" is visible on a display and corresponds to the ASCII codes from 0x21 to 0x7E.

Caution

If a value of c is not included in a range to be expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

islower

Character handling function

Judges whether a lower case letter or not.

Syntax

```
#include  < ctype.h >
int  islower ( int c );
     c;        /* character to be judged */
```

Return Value

non-0	: *c* is lower case letter
0	: *c* is anything other than lower case letter

Description

Returns any value other than 0 if *c* is a lower case letter and returns 0 if *c* is not a lower case letter. A lower case letter shall be defined as follows :

- Lower case 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

Caution

If a value of *c* is not included in a range to be expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

isprint

Character handling function

Judges whether a printable character including space.

Syntax

```
#include  < ctype.h >
int  isprint ( int  c );
     c;        /* character to be judged */
```

Return Value

non-0	: *c* is a printable character including space
0	: *c* is not a printable character which is not space

Description

Returns any value other than 0 if *c* is a printable character including space (' ') and returns 0 if *c* is not a printable character or a space. "Printable character including space" is visible on a display and corresponds to the ASCII codes from 0x20 to 0x7E.

Caution

If a value of *c* is not included in a range to be expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

ispunct

Character handling function

Judges whether a special character or not.

Syntax

```
#include  < ctype.h >
int  ispunct ( int c );
     c;        /* character to be judged */
```

Return Value

non-0	: *c* is special character
0	: *c* is anything other than special character

Description

Returns any value other than 0 if *c* is a special character and returns 0 if *c* is not a special character. A special character is any printable character excluding space, capital letter, small letter and decimal digit.

Caution

If a value of *c* is not included in a range to be expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

isspace

Character handling function

Judges whether a white-space or not.

Syntax

```
#include  < ctype.h >
int  isspace ( int c );
     c;        /* character to be judged */
```

Return Value

non-0	: *c* is a white-space
0	: *c* is anything other than a white-space

Description

Returns any value other than 0 if *c* is a white-space and returns 0 if c is not a white-space. A white-space shall be as the following characters :

- White-spaces white-space ( ), form feed (\f), line feed (\n), carriage return (\r), horizontal tab (\t) and vertical tab (\v)

Caution

If a value of *c* is not included in a range to be expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

isupper

Character handling function

Judges whether an upper case letter or not.

Syntax

```
#include  < ctype.h >
int  isupper ( int c );
     c;        /* character to be judged */
```

Return Value

non-0	: *c* is an upper case letter
0	: *c* is anything other than an upper case letter

Description

Returns any value other than 0 if *c* is an upper case letter and returns 0 if *c* is not an upper case letter. An upper case letter shall be defined as follows :

- Upper case 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

Caution

If a value of *c* is not included in a range to be expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

isxdigit

Character handling function

Judges whether a hexadecimal digit or not.

Syntax

```
#include  < ctype.h >
int  isxdigit ( int c );
     c;        /* character to be judged */
```

Return Value

non-0	: *c* is hexadecimal digit
0	: *c* is anything other than hexadecimal digit

Description

Returns any value other than 0 if *c* is a hexadecimal digit and returns 0 if *c* is not a hexadecimal digit.

- Hexadecimal digits :   0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

Caution

If a value of *c* is not included in a range to be expressed by the unsigned char type and if it is not EOF, the operation of this function is not guaranteed.

labs

General utility function

Obtains the absolute value of a long type integer.

Syntax

```
#include  < stdlib.h >
     long  labs ( long j );
     j;        /* integer calculating an absolute value */
```

Return Value

Absolute value of calculation result

Description

The labs function calculates an absolute value of a long type integer. If the calculation result cannot be expressed as a long type, the operation is not guaranteed.

ldexp

Mathematics function

Performs multiplication of a floating-point number and 2 to the *n*th power.

Syntax

```
#include  < math.h >
double  ldexp ( double x, int f );
     x;        /* floating-point number */
     f;        /* index (integer) */
```

Return Value

Result of $x*2^f$

Description

The ldexp function calculates a value of $x*2^f$ and return the result.

If the calculation result overflows, this function sets a value of ERANGE in errno and returns HUGE_VAL (-HUGE_VAL if negative) as a return value.

ldiv

General utility function

Divides a long type integer and obtains the quotient and remainder.

Syntax

```
#include  < stdlib.h >
ldiv_t  ldiv ( long number, long denom );
     number;        /* dividend */
     denom;         /* divisor */
```

Return Value

Quotient and remainder of the calculation result

Description

The ldiv function calculates the quotient and the remainder as the result of dividing number by *denom*. The return value is the ldiv_t structure and the quotient and the remainder are put in the quot and rem of the member, respectively.

localeconv

Localization function

Initialize struct lconv.

Syntax

```
#include  < locale.h >
struct lconv *localeconv ( void );
```

Return Value

Pointer to initialized `lconv` structure

Description

The localeconv function initializes the `lconv` structure

Caution

As the locale, only the "C" environment is supported.
In order to use the localeconv function, user-written _get_core or _rel_core function called by this function (internally) needs to be created.

localtime

Date and time function

Converts current calendar time to the local time.

Syntax

```
#include  < time.h >
struct tm *localtime ( const time_t *timer );
     timer;         /* time */
```

Return Value

Converted result (Pointer to the `tm` structure)

Description

The localtime function converts a calendar time shown by *time*r to the local time and converts it into the `tm` type.

Caution

As the locale, only the "C" environment is supported.
In order to use the localtime function, user-written getenv function called by this function (internally) needs to be created.

log

Mathematics function

Obtains natural logarithm of a floating-point number.

Syntax

```
#include  < math.h >
double  log ( double x );
     x;        /* floating-point number */
```

Return Value

Natural logarithm of *x*

Description

The log function calculates the natural logarithm of *x* and returns the result. If a value of *x* is negative, this function sets the value of EDOM in errno. If the value of *x* is 0.0, this function sets the value of ERANGE in errno.

log10

Mathematics function

Obtains the base 10 logarithm of a floating-point number.

Syntax

```
#include  < math.h >
double  log10 ( double x );
     x;             /* floating-point number */
```

Return Value

Common logarithm of *x*

Description

The logl0 function calculates the common logarithm of *x* and returns the result. If a value of *x* is negative, this function sets a value of EDOM in errno. If the value of *x* is 0.0, this function sets a value of ERANGE in errno.

longjmp

Non-local jump function

Recovers the execution environment saved by setjmp and transfers control to the program location of a setjmp call.

Syntax

```
#include  < setjmp.h >
void  longjmp ( jmp_buf env,  int ret );
     env;      /* variable in which executable environment
               is stored */
     ret;      /* return value to setjmp */
```

Return Value

None

Description

The longjmp function recovers the executable environment saved by the last call of setjmp function and transfers control to the location of program which called the setjmp function. Then, *ret* of the longjmp function returns as a return value of the setjmp function. The longjmp function does not return.

If longjmp function is executed before setjmp function is executed at a destination or if any function which called the setjmp function has already executed a return statement, the operation is not guaranteed.

Note

In using the longjmp function, the general-purpose registers are saved/recovered, by contrast, the accumulator (ACC) and the control registers are not saved/recovered.

malloc

General utility function

Allocates memory area.

Syntax

```
#include  < stdlib.h >
void  *malloc ( size_t size );
     size;          /* number of bytes of memory area
                    to be allocated */
```

Return Value

Top element address in the allocated memory area	:	Successful
NULL	:	Error (No memory area has been allocated. Any of the arguments is 0.)

Description

The malloc function allocates the memory area in bytes specified by *size*.

Note

12 bytes more area have been secured as memory area allocated by each the malloc functions. This memory area of 12 bytes is stored a allocation infomation (size etc.).

mblen

General utility function

Obtains the number of bytes composed of multibyte characters.

Syntax

```
#include  < stdlib.h >
int  mblen ( const char *s, size_t n );
     s;        /* pointer to the multibyte character */
     n;        /* number of the inspection bytes */
```

Return Value

- If *s* is NULL,

non-0	:	Multibyte character set depends on the status
0	:	Multibyte character set does not depend on the status.

- If *s* is not NULL,

number of bytes	:	*s* is a multibyte character. The number of bytes of it is returned.
0	:	The pointer *s* indicates a null character
-1	:	*s* is not a multibyte character

Description

The mblen function returns the number of bytes which consists of multibyte characters specified by *s*. If the mbtowc function is not affected by the shift status, mblen is the same value as the following example :

Example : `mbtowc((wchar_t*)0, s, n);`

Caution

The maximum value of multibyte-characte is 3 byte. And even “euc”, ”sjis”, ”utf8”, in addition to “C” are supported as the locale.

mbstowcs

General utility function

Converts a multibyte character string into a wide character string.

Syntax

```
#include  < stdlib.h >
size_t  mbstowcs ( wchar_t *pwcs, const char *s, size_t n );
     pwcs;     /* storage buffer of wide character string */
     s;        /* pointer to multibyte character string */
     n;        /* number of wide characters to be converted */
```

Return Value

Number of converted characters	:	Successful (However, a null character at the end shall not be included in number of characters)
-1	:	Casts -1 as size_t and returns it when any character other than multibyte character is detected during conversion

Description

The mbstowcs function converts the number of characters pointed to by *n* among the multibyte character string specified by *s* into the corresponding character string of wide characters and returns the result to the buffer *pwcs*.

Caution

The maximum value of multibyte-characte is 3 byte. And even “euc”, ”sjis”, ”utf8”, in addition to “C” are supported as the locale.

mbtowc

General utility function

Converts a multibyte character into a wide character.

Syntax

```
#include  < stdlib.h >
int  mbtowc ( wchar_t *pwc, const char *s, size_t n );
     pwc;      /* storage buffer of wide character */
     s;        /* pointer to multibyte character */
     n;        /* number of inspection bytes */
```

Return Value

Number of bytes	:	Both pwc and *s* are not NULL. The number of bytes of the converted multibyte characters.
0	:	*s* indicates a null character
-1	:	*s* does not correctly indicate a multibyte character

Note) If multibyte character encoding has state-dependent encoding, a non-zero is returned. Otherwise, a zero is returned.

Description

The mbtowc function processes multibyte characters specified by *s* as follows:

• When both *pwc* and *s* are not NULL :
It converts multibyte characters of *n* bytes from the point specified by *s* into wide characters, and stores them into the buffer *pws.*

• When *pwc* is NULL and *s* is not NULL :
It operates like the mblen function.

• When *s* is NULL, or both *pwc* and *s* are NULL :
It checks whether a multibyte character set depends on the status.

Caution

The maximum value of multibyte-characte is 3 byte. And even “euc”, ”sjis”, ”utf8”, in addition to “C” are supported as the locale.

memchr

String handling function

Locates, in a memory area, the position where a character first appears.

Syntax

```
#include  < string.h >
void *memchr ( const void *s, int c, size_t n );
     s;   /* pointer to memory area to be searched */
     c;   /* character to be searched */
     n;   /* number of characters to be searched */
```

Return Value

Pointer to character position	: As the result of search, the character has been found
NULL	: As the result of search, the character has not been found

Description

The memchr function returns the pointer, as a return value, to the position of the same character as the character *c* which appears for the first time in *n* characters from the beginning of the specified memory area.

memcmp

String handling function

Compares the contents of two memory areas.

Syntax

```
#include  < string.h >
int memcmp ( const void *s1, const void *s2, size_t n );
     s1;  /* pointer to compared memory area */
     s2;  /* pointer to comparing memory area */
     n;   /* number of characters of comparing memory
          location */
```

Return Value

Positive value	: content of *s1* > content of *s2*
0	: content of *s1* == content of *s2*
Negative value	: content of *s1* < content of *s2*

Description

The memcmp function compares the contents of the first *n* bytes of the memory areas specified by *s1* and *s2*.

memcpy

String handling function

Copies the contents of a memory area to the destination memory area.

Syntax

```
#include  < string.h >
void  *memcpy ( void *s1, const void *s2, size_t n );
     s1;  /* copy destination */
     s2;  /* original */
     n;   /* number of copied bytes */
```

Return Value

Pointer to the memory area at the copy destination

Description

The memcpy function copies *n* characters of the memory area *s2* to the destination memory area *s1*.

memmove

String handling function

Moves the contents from a memory area to the destination memory area.

Syntax

```
#include < string.h >
void  *memmove ( void *s1, const void *s2, size_t n );
     s1;  /* copy destination (storage buffer) */
     s2;  /* original */
     n;   /* number of copied bytes */
```

Return Value

s1

Description

The memmove function copies the first *n* bytes of content of *s2* to *s1*. The memmove function copies correctly even if the object (memory area) of *s1* and that of *s2* overlaps with each other (However, original data in the overlapped area is erased.). The operation is not guaranteed if the copied data exceeds the buffer area of *s1* as the result of copy.

memset

String handling function

Copies a character into the first *n* characters in memory area.

Syntax

```
#include  < string.h >
void  *memset ( void *s, int c, size_t n );
     s;   /* pointer to area in which character is set */
     c;   /* character to be set */
     n;   /* number of characters to be set */
```

Return Value

Pointer *s* to memory area in which character is set

Description

The memset function sets *n* characters of *c* into memory area pointed to by *s*.

mktime

Date and time function

Converts date and time (a struct tm) to the calendar time.

Syntax

```
#include  < time.h >
time_t  mktime ( struct tm *timeptr );
     timeptr;       /* pointer to tm structure */
```

Return Value

Conversion result	:	Converted into a calendar time correctly
-1	:	Could not be converted (Casts -1 to `time_t` type and returns it)

Description

The mktime function converts the broken-down time of the `tm` structure shown by *timeptr* into calendar time. The mktime function handles the `tm` structure specified by *timeptr* as follows :

- Ignores *tm_wday* of the `tm` type structure and the original value (upon reading) of *tm_yday*.
- Contents of the `tm` structure of *timeptr* are updated according to the calendar time.

modf

Mathematics function

Divides a floating-point number into integer and fractional parts.

Syntax

```
#include  < math.h >
double  modf ( double x, double *i );
     x;        /* floating-point number */
     i;        /* pointer to memory area which stores
               integral part */
```

Return Value

Fractional part of *x*

Description

The modf function divides the value of *x* into fractional part and integral part and returns the fractional part. It stores the integral part in the memory area specified by *i*.

perror

Input/output function

Outputs the error message corresponding to the error code to the standard error file (stderr).

Syntax

```
#include  < stdio.h >
void  perror ( const char *s );
     s;        /* pointer to error message */
```

Return Value

None

Description

The perror function outputs the error message indicated by *s* and the corresponding errno to a standard error file (stderr).

pow

Mathematics function

Obtains a floating-point number to *n*th power.

Syntax

```
#include  < math.h >
double  pow ( double x, double y );
     x, y;     /* floating-point number */
```

Return Value

Value of x^y : Successful
HUGE_VAL : Calculation result overflows

Description

The pow function returns a value of *x* raised to the power *y*. If the value of *x* is 0.0 and the value of *y* is 0.0 or less, or if a value of *x* is negative and a value of *y* is not an integer, this function sets EDOM in errno. If the calculation result overflows, this function sets ERANGE in errno.

printf

Input/output function

Converts data by following the format and outputs it to the standard output file (stdout).

Syntax

```
#include  < stdio.h >
int  printf ( const char *control ,… );
     control;       /* pointer to format string */
     …;             /* variable argument list (output data)
```

Return Value

Number of output characters : Successful
Negative value : Error

Description

The printf function converts and edits variable arguments according to format *control* and then outputs them to a standard output file (stdout). For the details of format *control*, refer to the description of the fprintf function.

putc

Input/output function

Outputs a character to a stream.

Syntax

```
#include  < stdio.h >
int  putc ( int c, FILE *fp );
     c;        /* character to be output */
     fp;       /* pointer to FILE structure */
```

Return Value

The character written	: Successful
EOF	: A write error occurs.

Description

The putc function (it may be implemented as a macro) outputs the character *c* into the stream specified by the file pointer *fp*.

When a write error occurs during the operation, the error indicator for the stream is set and this function returns EOF.

The error indicator can be referred by the ferror function.

putchar

Input/output function

Outputs a character to the standard output file (stdout).

Syntax

```
#include  < stdio.h >
int  putchar ( int c );
     c;   /* character to be output */
```

Return Value

The character written	: Successful
EOF	: A write error occurs.

Description

The putchar function (it may be implemented as a macro) is the same as `putc (c, stdout).`

When a write error occurs during the operation, the error indicator for the stream is set and this function returns EOF.

The error indicator can be referred by the ferror function.

puts

Input/output function

Outputs a string to the standard output file (stdout).

Syntax

```
#include  < stdio.h >
int  puts ( char *s );
     s;   /    * pointer to string to be output */
```

Return Value

0	: Successful
EOF	: A write error occurs.

Description

The puts function outputs the string indicated by *s* to the standard output (stdout). The null character at end of the string is replaced with the new-line character upon output.

When a write error occurs during the operation, the error indicator for the stream is set and this function returns EOF.

The error indicator can be referred by the ferror function.

qsort

General utility function

Performs sorting.

Syntax

```
#include  < stdlib.h >
void  qsort ( const void *base, size_t nmemb, size_t size,
              int(*compar)
                 (const void *,
                  const void *)
             );
     base;     /* pointer to the sort target table */
     nmemb;    /* the number of members to be sorted */
     size;     /* size (bytes) of the member to be sorted */
     compar;   /* pointer to the function to be compared */
```

Return Value

None

Description

The qsort function sorts the table denoted by *base*. The order of the sorted data depends on the compar function. The specification of this function is the same as that for the bsearch function.

raise

Signal handling function

Send a signal to the executing program.

Syntax

```
#include  < signal.h >
int  raise ( int sig );
     sig;      /* signal to be generated */
```

Return Value

0	: Successful
non-0	: Error

Description

The raise function sends the signal indicated by *sig* to the program.

Caution

To call the raise function in actual, the user-written raise function is required (see Section 11.1 and Table 11.2).

rand

General utility function

Generates a pseudo-random integer which resides between 0 and RAND_MAX.

Syntax

```
#include  < stdlib.h >
int  rand ( void );
```

Return Value

Generated pseudo-random integer

Description

The rand function generates a pseudo-random integer in the range of 0 to RAND_MAX.

realloc

General utility function

Changes the memory area size to the specified size.

Syntax

```
#include  < stdlib.h >
void  *realloc ( void *ptr, size_t size );
     ptr;      /* pointer to the memory area to be changed */
     size;     /* size (bytes) of the changed memory area */
```

Return Value

The start address of the allocated memory area	: Memory area is allocated.
NULL	: Memory space is not allocated, or successfully freed. (*size* is 0, *ptr* is not 0.)

Description

The realloc function changes the size of the memory space indicated by *ptr* to the bytes denoted by the *size*. If the changed memory size is smaller than before, the contents is unchanged up to the changed memory size.

If the space denoted by *ptr* of the realloc function is not allocated by the calloc, malloc or realloc function, or the space is already freed by the free or realloc function, the operation is not guaranteed.

When *ptr* is NULL, the realloc function operates like the malloc function. When *size* is 0 and *ptr* is not NULL, the memory area is freed.

Note

12 bytes more area have been secured as memory area allocated by each the realloc functions. This memory area of 12 bytes is stored a allocation infomation (size etc.).

remove

Input/output function

Deletes a file.

Syntax

```
#include  < stdio.h >
int  remove ( const char *filename );
     filename;      /* name of the file to be deleted */
```

Return Value

0	: Successfully deleted
non-0	: Error

Description

The remove function deletes the file denoted by *filename* even if the denoted file is currently open.

Caution

Whether a currently open file is changed or deleted depends on the operating environment.

To call the remove function in actual, the user-written remove function is required (see Section 11.1 and Table 11.2).

rename

Input/output function

Renames a file.

Syntax

```
#include  < stdio.h >
int  rename ( const char *old, const char *new );
     old;      /* original file name */
     new;      /* new file name */
```

Return Value

0	: File name is changed
non- 0	: Error

Description

The rename function changes the name of the file specified by *old* to the name denoted by *new*. The rename function will change the name of the file which is denoted by *old* and currently open, also it will change the name of the file specified by *new* even if the name specified already exists.

Caution

Whether the currently open file is changed or deleted depends on the operating environment.

To call the rename function in actual, the user-written rename function is required (see Section 11.1 and Table 11.2).

rewind

Input/output function

Moves the current read/write position on a stream to the beginning of the file.

Syntax

```
#include  < stdio.h >
void  rewind ( FILE *fp );
     fp;       /* pointer to FILE structure */
```

Return Value

None

Description

The rewind function moves the current read/write position of the I/O stream denoted by the file pointer *fp* to the beginning of the file. The rewind function disables the ungetc function.

scanf

Input/output function

Gets data from the standard input file (stdin) and converts the data by following the format.

Syntax

```
#include  < stdio.h >
int  scanf ( const char *control, ... );
     control;  /* pointer to format string */
     ...;      /* variable argument list (receivers) */
```

Return Value

The number of arguments converted and substituted	:	Successful (the number excludes arguments for `%n` conversion and arguments inhibited by `*`)
EOF	:	The end of the input data is reached before the completion of the first conversion (except for `%%`), or an error occurs.

Description

The scanf function gets data from the standard input file (stdin), exchanges and edits it according to the format specified by *control*, and stores the results into the memory area specified by the variable argument list (…) .

For further information on format *control*, see the fscanf function description.

setbuf

Input/output function

Defines a buffer for an I/O stream.

Syntax

```
#include  < stdio.h >
void  setbuf ( FILE *fp, char *buf );
     fp;       /* pointer to FILE structure */
     buf;      /* pointer to buffer */
```

Return Value

None

Description

The setbuf function changes the buffer for the stream associated with the input/output file pointed to by *fp* from the default buffer[Note1] to the storage area denoted to by *buf*. The buffer size is unchanged. To change the buffer size, use the setvbuf function.

Setting NULL in *buf* causes the buffer not to be used (effects no buffering). This is equivalent to specifying _IONBF for the third argument *type* (buffer management method) of the setvbuf function.

Note1) The default buffer is an automatically allocated buffer when either the fopen function or the freopen function opens a file. Its buffer size (BUFSIZ) is defined in the header file stdio.h.

setjmp

Non-local jump function

Saves the current environment to a memory area.

Syntax

```
#include  < setjmp.h >
int  setjmp ( jmp_buf env  );
     env;      /* the current environment */
```

Return Value

0	: The setjmp function has been executed. The current environment has been set to *env*.
The second argument *ret* of longjmp function	: Control is returned to the setjmp function upon execution of the longjmp. If the second argument *ret* of the longjmp function is 0, 1 is returned to setjmp.

Description

The setjmp function saves the current calling environment in *env* into the specified memory area. By using this function in conjunction with the longjmp function, a global jump to outside the function can be achieved. This function is used to pass the error handling execution control to the previously called routine without using the normal function calling or return convention.

When the setjmp function is called with a complex expression, a portion of current execution environment such as the temporary result of an expression evaluation may be lost. To avoid this, use the setjmp function only for the purpose of comparing the result of the setjmp function and a constant expression, and don't call this function in a complex expression.

Note

In using the setjmp function, the general-purpose registers are saved/recovered, by contrast, the accumulator (ACC) and the control registers are not saved/recovered.

setlocale

Localization function

Sets and searches for locale information.

Syntax

```
#include  < locale.h >
char  *setlocale ( int category,  const char *locale );
     category;      /* portion of locale to be set/search */
     locale;        /* locale to be set */
```

Return Value

The pointer to the available string for locale	:	Successful (*locale* will not be returned)
NULL	:	Both *category* and *locale* are invalid

Description

As the locale, only the "C" environment is supported.
The setlocale function sets the portion of the program's locale denoted by *category* into the locale specified by *locale*. If *locale* is NULL, the setlocale function searches the current locale information and the locale is not changed.
The following are the macros that can be specified in *category* :

category	**Description**
LC_ALL	Sets or searches for all locale information.
LC_COLLATE	Sets or searches for information that affects the strcoll and strxfrm functions.
LC_CTYPE	Sets or searches for information that affects all character handling functions and multibyte handling functions except for the isdigit and isxdigit functions.
LC_MONETARY	Sets or searches for information that affects the currency information which the localeconv function returns.
LC_NUMERIC	Sets or searches for information that affects input and output functions, decimal points used by character handling functions, and information, excluding the currency information which the localeconv function returns.
LC_TIME	Sets or searches for information that affects the strftime function.

setvbuf

Input/output function

Defines and sets a buffer for an I/O stream.

Syntax

```
#include  < stdio.h >
int  setvbuf ( FILE *fp, char *buf, int type, size_t size  );
     fp;             /* pointer to FILE structure */
     buf;            /* pointer to buffer */
     type;           /* buffer management method */
     size;           /* size of buffer */
```

Return Value

0	: New buffer has been defined
non-0	: Error

Description

The setvbuf function changes the buffer for the stream associated with the input/output (I/O) file pointed to by *fp* from the default buffer[Note1)] to the storage area denoted to by *buf*. And it redefines the buffer size to *size* and the buffer management method to *type*. The following three buffer management methods are available:

type	**Description**
_IOFBF	Full buffering. Every I/O process uses a buffer. Data are taken out of the buffer either when the buffer is fulled or when flushed.
_IOLBF	Line buffering. Every I/O process uses a buffer line by line. Data are taken out of the buffer either when the buffer is fulled, when flushed, or when a newline character is encountered.
_IONBF	No buffering. I/O processing is done in quantities of reading/ writing from/to a stream without using the buffer.

Caution

The setvbuf function must be used after the stream input/output file is opened and before the start of the input/output process. Do not deallocate the buffer before closing the file.

[Note1)] The default buffer is an automatically allocated buffer when either the fopen function or the freopen function opens a file. Its buffer size (BUFSIZ) is defined in the header file stdio.h.

signal

Signal handling function

Sets up a signal handler that responds to the signal.

Syntax

```
#include  < signal.h >
void  ( *signal ( int sig, void (*func)(int) ) )(int);
     sig;      /* signal number */
     func;     /* The handler to be executed upon detecting
               the signal */
```

Return Value

The latest func value	: Successful
SIG_ERR	: Error (returns corresponding value to errno)

Description

The signal function calls the process handler denoted by *func* when the signal specified by *sig* is detected.

func specifies the following two special macros in addition to the user defined ones :

- SIG_DFL Executes the default signal process
- SIG_IGN Ignores signals

Caution

To call the signal function in actual, the user-written signal function is required (see Section 11.1 and Table 11.2).

sin

Mathematics function

Obtains the sine of the radians of a floating-point number.

Syntax

```
#include  < math.h >
double  sin ( double x );
     x;            /* floating-point number (in radians) */
```

Return Value

Calculated argument value

Description

The sin function calculates sine of *x*.

sinh

Mathematics function

Obtains hyperbolic sine of a floating-point number.

Syntax

```
#include  < math.h >
double  sinh ( double x );
     x;            /* floating-point number */
```

Return Value

Calculated argument value

Description

The sinh function calculates the hyperbolic sine of *x*. If the calculation result cannot be expressed in double type value, the function sets the ERANGE value in the errno. If the calculation result overflows, the function returns HUGE_VAL (-HUGE_VAL if negative).

sprintf

Input/output function

Converts the data by following the format and outputs the data to an area.

Syntax

```
#include  < stdio.h >
int  sprintf ( char *s, const char *control , ...  );
     s;        /* pointer to the memory area to which
                  data is output */
     control;  /* pointer to format string */
     ...;      /* variable argument list (output data) */
```

Return Value

Number of output characters

Note) No return value when an error is shown

Description

The sprintf function converts and edits the output data specified by the variable argument list according to the format *control* and then outputs the result to the memory area denoted by *s*. For more information on the format *control,* refer to the description of the fprintf function.

sqrt

Mathematics function

Obtains the positive square root of a floating-point number.

Syntax

```
#include  < math.h >
double  sqrt ( double x );
     x;        /* floating-point number */
```

Return Value

Positive square root of *x*

Description

The sqrt function obtains the positive square root of *x*. If *x* is a negative, the value of EDOM is set into `errno`.

srand

General utility function

Sets the initial values of the pseudo-random integers which the rand function generates.

Syntax

```
#include  < stdlib.h >
void  srand ( unsigned int seed );
     seed;     /* initial value for a sequence of pseudo-
                  random integers generation */
```

Return Value

None

Description

The srand function sets the initial value (seed) for the sequence of pseudo-random integers generated by the rand function. If the srand function sets the same value as the initial value while the rand function is generating pseudo-random numbers, the same sequence of pseudo-random integers is generated again.

When the rand function is called before the srand function, the initial value for the sequence of pseudo-random integer generation is set to 1.

sscanf

Input/output function

Gets data from a memory area and converts the data by following the format.

Syntax

```
#include  < stdio.h >
int  sscanf ( char s, const char *control, ...  );
     s;             /* pointer to the memory area to which
                       data is to be stored */
     control;       /* pointer to format string */
     ...;           /* variable argument list (receivers) */
```

Return Value

The number of arguments converted and substituted	:	Successful (the number excludes arguments of `%n` conversion and arguments inhibited by `*`)
EOF	:	The end of input data is reached before completion of the first conversion (except for `%%`), or an error occurs

Description

The sscanf function gets data from the memory area denoted by *s*, exchanges and edits the data in the format specified by *control*, and stores the result into the memory area denoted by the variable argument list (,...) . For further information on the format *control*, refer to the fscanf function description.

strcat

String handling function

Links a string to the end of a string.

Syntax

```
#include  < string.h >
char  *strcat ( char *s1, const char *s2 );
     s1;       /* pointer to string */
     s2;       /* pointer to string to be appended to s1 */
```

Return Value

The pointer, *s1*, after linking

Description

The strcat function links the string *s2* to the end of the string *s1*. The last NULL character of string *s1* is replaced with the first character of *s2* string. A NULL character is always added to the end of the linked *s2* string. The strncat function can specify the number of characters to be linked to, by using the parameter *n*.

strchr

String handling function

Locates, in a string, the position where a character first appears.

Syntax

```
#include  < string.h >
char  *strchr ( const char *s, int c );
     s;        /* pointer to string used to search */
     c;        /* character to be searched */
```

Return Value

Pointer to the first searched character	: Character is found
NULL	: Character is not found

Description

The strchr function returns the pointer to where the character *c* first appears in the string *s*.

strcmp

String handling function

Compares two strings.

Syntax

```
#include  < string.h >
int  strcmp ( const void *s1,  const void *s2 );
     s1;  /* pointer to string to be compared */
     s2;  /* pointer to string to be used for comparison */
```

Return Value

Positive value	: contents of s1 > contents of s2
0	: contents of s1 == contents of s2
Negative value	: contents of s1 < contents of s2

Description

The strcmp function compares the contents of string specified by *s1* and those specified by *s2* and returns the result.

strcoll

String handling function

Compares the two strings based on the current locale.

Syntax

```
#include  < string.h >
int  strcoll ( const char *s1, const char *s2 );
     s1;       /* comparison string 1 */
     s2;       /* comparison string 2 */
```

Return Value

Positive value	: contents of *s1* > contents of *s2*
0	: contents of *s1* == contents of *s2*
Negative value	: contents of *s1* < contents of *s2*

Description

As the locale, only the "C" environment is supported.

The strcoll function compares *s1* with *s2* based on LC_COLLATE (locale.h definition) denoted by the current locale .

strcpy

String handling function

Copies the contents (including null characters) of the source string to the target memory area.

Syntax

```
#include  < string.h >
char  *strcpy ( char *s1, const char *s2 );
     s1;     /* pointer to space to which string is copied */
     s2;     /* pointer to string to be copied */
```

Return Value

Pointer of the memory area to which the characters are copied

Description

The strcpy function copies the string including the terminating null character specified by *s2* into the memory area specified by *s1*.

strcspn

String handling function

Computes the length of initial segment of a string which consists of unspecified characters.

Syntax

```
#include  < string.h >
size_t  strcspn ( const char *s1, const char *s2 );
     s1;  /* pointer to string to be checked */
     s2;  /* pointer to string used to check the s1 string */
```

Return Value

The length of the searched string

Description

The strcspn function computes the length of the maximum initial segment of the string pointed to by *s1* which consists of characters not from the string pointed to by *s2*. And it returns the length (i.e., the number of characters).

strerror

String handling function

Returns the error message.

Syntax

```
#include  < string.h >
char  *strerror ( int e );
     e;        /* error number */
```

Return Value

Pointer to the error message corresponding to the error number *e*

Description

The strerror function returns the pointer to the error message corresponding to the error number *e* as the return value.

strftime

Date and time function

Converts date and time (a struct tm) to the format specified.

Syntax

```
#include < time.h >
size_t  strftime (  char *s, size_t maxsize,
                    const char *format,
                    const struct tm *timeptr );
     s;        /* buffer for storing converted string */
     maxsize;  /* maximum number of conversion characters */
     format;   /* format string */
     timeptr;  /* pointer to tm structure */
```

Return Value

Number of characters written	: Number of characters written into *s* is smaller than or equal to maxsize
0	: Number of characters written into *s* is larger than the maxsize (contents is unknown)

Description

As the locale, only the "C" environment is supported.

The strftime function converts the broken-down time of the `tm` structure specified by *timeptr* according to the format specified by *format* and stores the conversion result into *s*. The maximum number of characters to be stored into *s* is the number specified by *maxsize*.

The following lists the conversion specifiers that can be used in *format*. Any conversion specifier not included in the table makes operation result unknown.

(1/2)

Conversion Specifier	Description
%a	Abbreviation for the day of the week in the locale
%A	The name of the day of the week in the locale
%b	Abbreviation for the month in the locale
%B	The name of month in the locale
%c	Representation of the date and time appropriate in the locale
%d	Representation of the date of a month (integer, 01 through 31)

(2/2)

Conversion Specifier	**Description**
%H	Representation of the time of a day in 24-hour system (integer, 00 through 23)
%I	Representation of the time of a day in 12-hour system (integer, 01 through 12)
%j	The number of days starting at the first day of the year (integer, 001 through 366)
%m	Representation of the month (integer, 01 through 12)
%M	Representation of the minute (integer, 00 through 59)
%p	AM or PM of the locale used when representing in 12-hour system
%S	Representation of the second (integer, 00 through 59)
%U	The number of the week starting at the first week of the year (integer, 00 through 53 with Sunday defined as the first day of week)
%w	The *n*th day of the week with Sunday defined as the 0th day (integer, 0 through 6)
%W	The number of weeks starting at the first week of the year (integer, 00 through 53 with Monday defined as the first day of week)
%x	Representation of the date appropriate in the locale
%X	Representation of the time of a day appropriate in the locale
%y	Representation of *n*th year of a century (integer, 00 through 99)
%Y	Representation of a year of the Christian era (integer)
%Z	Time zone or abbreviation of the zone (null character if the time zone cannot be specified)
%%	Representation of % itself

strlen

String handling function

Measures the size of string.

Syntax

```
#include  < string.h >
size_t strlen ( const char *s );
     s;        /* pointer to string whose length is
                  to be measured */
```

Return Value

Number of characters of the string measured

Description

The strlen function measures the length of the string *s*. The null character indicating the end of the character is not counted.

strncat

String handling function

Links the specified number of characters to the string.

Syntax

```
#include  < string.h >
char *strncat ( char *s1, const char *s2, size_t n );
     s1;       /* pointer to string to be linked by */
     s2;       /* pointer to string to be linked to */
     n;        /* number of characters to be linked to */
```

Return Value

Pointer *s1* of the linked string

Description

The strncat function links the string *s2* to the end of the string *s1*. The last null character of the string *s1* is replaced with the first character of *s2*. The last character of the linked string *s2* is always followed by a null character. The number of characters to be linked is specified by the parameter *n*.

strncmp

String handling function

Compares specified number of characters of two strings.

Syntax

```
#include  < string.h >
int  strncmp ( const void *s1, const void *s2, size_t n );
     s1;  /* pointer to string to be compared */
     s2;  /* pointer to string used to compare */
     n;   /* maximum number of characters to be compared */
```

Return Value

Positive value	: contents of *s1* > contents of *s2*
0	: contents of *s1* == contents of *s2*
Negative value	: contents of *s1* < contents of *s2*

Description

The strncmp function compares the contents of string specified by *s1* with the contents of the string specified by *s2*, up to the nth characters specified by the parameter *n*, and sets the result as the return value.

strncpy

String handling function

Copies the specified number of characters from the string to memory.

Syntax

```
#include  < string.h >
char  *strncpy ( char *s1, const char *s2, size_t n );
     s1;       /* pointer to space to which copy is made */
     s2;       /* pointer to string to be copied */
     n;        /* the number of characters to be copied */
```

Return Value

Pointer to the memory area to which the characters are to be copied

Description

The strncpy function copies the specified number of characters within the string *s2* to the specified memory area *s1*. If the size of the string specified by *s2* is smaller than the number of characters specified by *n,* null characters are added until the *s2*-specified character size matches the *n*-specified character size. If the number of *s2*-specified characters is larger than that of the *n*-specified character size, the last character of the string copied to *s1* is other than a null character.

strpbrk

String handling function

Locates the position where one of the specified characters first appears in a string.

Syntax

```
#include  < string.h >
char  *strpbrk ( const char *s1, const char *s2 );
     s1;       /* pointer to string in which
                  the character is searched */
     s2;       /* pointer to string which consists of
                  characters to be found */
```

Return Value

Pointer to the character found	: The character is found.
NULL	: The character is not found.

Description

The strpbrk function locates in the string *s1* the position where one of characters in the string *s2* first appears and returns the pointer to that position.

strrchr

String handling function

Locates the position where a character last appears in a string.

Syntax

```
#include  < string.h >
char  *strrchr  ( const char *s, int c );
     s;        /* pointer to string in which
                  the character is searched */
     c;        /* the character to be searched for */
```

Return Value

Pointer to the character found	: The character is found.
NULL	: The character is not found.

Description

The strrchr function searches for the character *c* in the character string *s* and locates the location where the *c* last appears, and returns the pointers of the position. The null character indicating the end of the string *s* is included in the characters to be searched.

strspn

String handling function

Computes the length of initial segment of a string which consists of specified characters.

Syntax

```
#include  < string.h >
     size_t  strspn ( const char *s1, const char *s2 );
     s1;       /* pointer to string to be checked */
     s2;       /* pointer to string used to check s1 */
```

Return Value

The length of the searched string

Description

The strspn function searches the string *s1* starting from the beginning for a series of characters which match the characters of the string *s2* and returns the number of continuous characters that precede the first match series of characters of s2.

strstr

String handling function

Finds the first occurrence point of a string within another.

Syntax

```
#include  < string.h >
char  *strstr ( const char * s1, const char *s2 );
     s1;       /* pointer to string to be searched */
     s2;       /* pointer to string to be searched for */
```

Return Value

Pointer to the character found	: The character is found in the string *s1*.
NULL	: The character is not found, or *s2* is null.

Description

The strstr function locates in the character string *s1* the position where the character string *s2* first appears and returns the pointer to that position. If *s2* is null, the return value is also null.

strtod

General utility function

Converts a string into a double type floating-point number.

Syntax

```
#include  < stdlib.h >
double  strtod ( const char *nptr, char **endptr );
     nptr;          /* pointer to string to be converted */
     endptr;        /* end point of reading */
```

Return Value

Converted value

Description

The strtod function converts a string into a double type numeric value. The input string must be an array of characters that, after conversion into a numeric value of a type, can be interpreted as a valid numeric value of that type. The strtod function, upon encountering a character which cannot be interpreted as a value, stops inputting current string and sets the pointer to the first character found in the input string which cannot be interpreted as a numeric value, in the area indicated by *endptr*. If the *endptr* is null, the function does not perform this setting.

If the conversion results in overflow or underflow, the function sets ERANGE value to the errno and returns either HUGE_VAL (-HUGE_VAL in case of negative value) or 0.

strtok

String handling function

Separates a string into tokens.

Syntax

```
#include  < string.h >
char  *strtok ( char *s1, const char *s2 );
     s1;  /* pointer to string to be divided into tokens */
     s2;  /* pointer to string used to divide a string */
```

Return Value

Pointer to the top character of the token after division	: Divided into tokens
NULL	: Cannot be divided into tokens

Description

The strtok function uses a particular character in the string *s2* as the punctuator to divide the string *s1* into tokens. The function is called continuously, once for each token. The function, upon first calling, separates the first group of the *s1* string by using the character in the string s2; and the second group upon the second calling, etc.

When the second and subsequent calls, specify NULL as the first parameter. The contents of *s2* may be different from call to call. A null character is added to end of each token.

strtol

General utility function

Converts a string into a long type integer.

Syntax

```
#include  < stdlib.h >
long  strtol ( const char *nptr, char **endptr, int base );
     nptr;      /* pointer to string to be converted */
     endptr;    /* end point of reading */
     base;      /* radix of conversion (0,2-36) */
```

Return Value

Numeric value after conversion	: Successful
0	: Failed

Description

The strtol function reads a string pointed to by *ntpr*, and converts it into a value (long type) having a radix specified by *base*.

The strtol function reads a string to be brought to conversion sequentially from the first character and stops reading when it detects either a null character or a character that it cannot interpret as a number that assumes a value having the radix *base*; and sets in *endptr* the pointer pointing to the character handled at the time (only when *endptr* does not hold NULL). Space characters preceding the string to be brought to conversion (the first segment where a non-space character does not appear) are ignored and not converted.

If *base* is 0, strtol makes a judgment on the radix from the first character that *nptr* points to (see 4.1.3.2 "Integer Constants"). When the value of *base* is between 2 and 36, the value of *base* becomes the radix for conversion.

Characters a through z (and A through Z) among a string to be brought to conversion are made to correspond to 10 through 35, and a character equal to or greater than the value of *base* is recognized as a character that strtol cannot interpret. 0 after a sign and 0x (0X) at the time when *base* is 16 are ignored.

If conversion fails, strtol returns 0 as a return value, and *nptr* is set in *endptr* (only when *endptr* does not hold NULL).

If the value after conversion overflows (i.e., it cannot be expressed by long), strtol function returns LONG_MAX (LONG_MIN if negative) as a return value, and ERANGE is set in `errno`.

strtoul

General utility function

Converts a string into an unsigned long type integer.

Syntax

```
#include  < stdlib.h >
unsigned long int  strtoul (  const char *nptr,
                              char **endptr,
                              int base );
     nptr;      /* pointer to string to be converted */
     endptr;    /* end point of reading  */
     base;      /* radix */
```

Return Value

Numeric value after conversion	: Successful
0	: Failed

Description

The strtoul function reads a string pointed to by *ntpr*, and converts it into a value (unsigned long type) having a radix specified by *base*.

The strtoul function reads a string to be brought to conversion sequentially from the first character and stops reading when it detects either a null character or a character that it cannot interpret as a number that assumes a value having the radix *base*; and sets in *endptr* the pointer pointing to the character handled at the time (only when *endptr* does not hold NULL). Space characters preceding the string to be brought to conversion (the first segment where a non-space character does not appear) are ignored and not converted.

If *base* is 0, strtoul function makes a judgment on the radix from the first character that *nptr* points to (see 4.1.3.2 “Integer Constants”). When the value of *base* is between 2 and 36, the value of *base* becomes the radix for conversion.

Characters a through z (and A through Z) among a string to be brought to conversion are made to correspond to 10 through 35, and a character equal to or greater than the value of *base* is recognized as a character that strtoul cannot interpret. 0 after a sign and 0x (0X) at the time when *base* is 16 are ignored.

If conversion fails, strtoul function returns 0 as a return value, and *nptr* is set in *endptr* (only when *endptr* does not hold NULL).

If the value after conversion overflows (i.e., it cannot be expressed by unsigned long), strtol returns ULONG_MAX as a return value, and ERANGE is set in `errno`.

strxfrm

String handling function

Converts the string based on the current locale.

Syntax

```
#include  < string.h >
size_t  strxfrm ( char *s1, const char *s2, size_t n );
     s1;       /* buffer to store the converted string */
     s2;       /* pointer to string to be converted */
     n;        /* number of characters converted */
```

Return Value

The number of bytes of the converted string (except the last null of the string)

Note) When the return value is larger than the *n*, the contents of *s1* are unknown.

Description

As the locale, only the "C" environment is supported.

The strxfrm function converts *n* bytes of the *s2* string starting at the beginning of the string according to LC_COLLATE specified by the current locale and stores the result into *s1*.

system

General utility function

Passes a command string to the host environment.

Syntax

```
#include  < stdlib.h >
int  system ( const char *string );
     string;        /* command string */
```

Return Value

non-0	: *string* is NULL and command processor exists
0	: *string* is NULL and no command processor exists

Note) If the command string is not NULL, the return value depends on the processing system.

Description

The system function passes the string specified by *string* to the host environment where the command processor runs. When NULL is specified as *string*, the function checks whether the command processor exists.

Caution

To call the system function in actual, the user-written system function is required (see Section 11.1 and Table 11.2).

tan

Mathematics function

Obtains the tangent of the radians of a floating-point number.

Syntax

```
#include  < math.h >
double  tan ( double x );
     x;         /* floating-point number (in radians) */
```

Return Value

Calculated parameter, the tangent of *x*

Description

The tan function calculates the tangent of *x*. If the result of the function cannot be expressed as a double type value, the ERANGE value is set into errno. If the operation results in an overflow, HUGE_VAL (–HUGE_VAL in the case of negative value) is returned.

tanh

Mathematics function

Obtains hyperbolic tangent of a floating-point number.

Syntax

```
#include  < math.h >
double  tanh ( double x );
     x;                   /* floating-point number */
```

Return Value

Calculated parameter

Description

The tanh function calculates the hyperbolic tangent of *x*. If the calculation result overflows, the function returns HUGE_VAL (-HUGE_VAL if negative value).

time

Date and time function

Reads the current calendar time.

Syntax

```
#include  < time.h >
time_t  time ( time_t *timer );
     timer;          /*  calendar time  */
```

Return Value

Current calendar time	: The timer is not NULL
-1	: Current calendar time was not obtained (casting -1 to time_t type upon returning)

Description

The time function writes the current calendar time into the buffer specified by *timer*.

Caution

To call the time function in actual, the user-written time function is required (see Section 11.1 and Table 11.2).

tmpfile

Input/output function

Creates a temporary file.

Syntax

```
#include  < stdio.h >
FILE  *tmpfile ( void );
```

Return Value

Pointer to the stream of the generated file	: The file is generated
NULL	: The file cannot be generated

Description

The tmpfile function generates a temporary file. The generated file is opened in the binary file update mode (wb+) and is automatically deleted upon closing of the file or terminating the program.

tmpnam

Input/output function

Creates a not-existing temporary file name.

Syntax

```
#include  < stdio.h >
char  *tmpnam ( char *s );
     s;              /* buffer to store the name of file
                        generated by the tmpnam */
```

Return Value

Generated file name	: The file name is generated
NULL	: The number of the tmpnam function calls exceeds TMP_MAX

Description

The tmpnam function generates a new temporary file name and stores it into the buffer specified by *s*. When NULL is substituted for *s,* the result is written into the tmpnam internal buffer. The buffer is overwritten every time tmpnam is called.

The tmpnam function can be repeatedly called up for TMP_MAX times (defined in stdio.h). The tmpnam function returns NULL upon exceeding TMP_MAX times.

At least the space whose size is equal to L_tmpnam (defined in stdio.h) must be secured in the buffer.

tolower

Character handling function

Converts an upper case letter into lower case.

Syntax

```
#include  < ctype.h >
int  tolower ( int c );
     c;             /* character to be converted */
```

Return Value

Converted character	: When the character *c* specified as the parameter meets the condition
Character *c* (as it is)	: When the character *c* specified as the parameter cannot meet the condition

Description

The tolower function converts the character corresponding to the parameter *c* into lower case if it is an upper case character, and returns the lower case character. Otherwise, returns the character without converting it.

toupper

Character handling function

Converts a lower case letter into upper case.

Syntax

```
#include  < ctype.h >
int  toupper ( int c );
     c;            /* character to be converted */
```

Return Value

Converted character	: When the character *c* specified as the parameter meets the condition
Character *c* (as it is)	: When the character *c* specified as the parameter cannot meet the condition

Description

The toupper function converts the character corresponding to the parameter *c* into upper case if it is a lower case character, and returns the upper case character. Otherwise, returns the character without converting it.

ungetc

Input/output function

Returns a character a stream.

Syntax

```
#include  < stdio.h >
int  ungetc ( int c, FILE *fp );
     c;         /* character of pushback */
     fp;        /* pointer to FILE structure */
```

Return Value

Pushed-back character *c*	: Successful (Normal)
EOF	: Failed

Description

The ungetc function returns the character *c* to the I/O stream specified by the file pointer *fp*. The pushed-back character will become the next input data if the fflush, fseek or rewind function is not called. If the ungetc function is called twice or more without first one of the three functions being called, the operation is not guaranteed.

Upon execution of the ungetc function, the position indicator for the file is moved backward by one step. If the indicator is already at the beginning of the file, the indicator is not guaranteed.

When an error occurs during the operation, the error indicator for the stream is set or the EOF indicator is set; and this function returns EOF.

The EOF indicator can be referred by the ferror function. The error indicator can be referred by the ferror function.

va_arg (Macro)

Variable arguments access function

Gets variable arguments in turn.

Syntax

```
#include  < stdarg.h >
type  va_arg ( va_list ap, type );
     ap;       /* pointer to variable parameter list
                  (the same as ap initialized by va_start) */
     type;     /* type of return value */
```

Return Value

Value of the argument

Description

By using macros va_start, va_arg and va_end, the arguments of the function having variable parameters can be referenced.

Specify the va_list type *ap* initialized by the va_start macro as the first parameter. The second parameter *type* is a the type name of parameter to be referenced. Each call of the va_arg function updates the value of *ap* so that the values of successive arguments are returned as the return value one by one.

Caution

When the *type* is a type whose size changes by type conversion, the parameter is not correctly referenced. The operation is not guaranteed if such a type is specified.

va_end (Macro)

Variable arguments access function

Ends the reference to variable arguments.

Syntax

```
#include  < stdarg.h >
void  va_end ( va_list ap );
     ap;       /* pointer to variable parameter list
                  (the same as ap initialized by va_start) */
```

Return Value

None

Description

By using macros va_start, va_arg and va_end, the arguments of the function having variable parameters can be referenced.

The va_end macro terminates the reference to variable arguments. *ap* must be equal to the *ap* initialized by the va_start macro. If the va_end macro is not called before the return, the operation is not guaranteed.

va_start (Macro)

Variable arguments access function

Initializes to reference variable arguments.

Syntax

```
#include  < stdarg.h >
void  va_start ( va_list ap, parmN );
     ap;            /* pointer to variable parameter list */
     parmN;         /* the rightmost identifier in
                       parameter list */
```

Return Value

None

Description

By using macros va_start, va_arg and va_end, the arguments of the function having variable parameters can be referenced.

The va_start macro initializes the *ap* used by the va_arg and va_end macro. The *parmN* is the identifier indicating the rightmost parameter in the parameter list of the external function definition, namely, the identifier just before the variable parameter list (,...).

To reference an unnamed parameter of the variable parameters, first call the va_start macro.

vfprintf

Input/output function

Outputs a variable argument list to a stream by following the format.

Syntax

```
#include  < stdarg.h >
#include  < stdio.h >
int  vfprintf ( FILE *fp, const char *control, va_list arg );
     fp;            /* pointer to FILE structure */
     control;       /* pointer to format string */
     arg;           /* pointer to variable argument list */
```

Return Value

Number of characters output	: Successful
Negative value	: Output error occurs

Description

The vfprintf function converts and edits the variable arguments pointed to by *arg* according to the format *control* and then outputs the result to the stream specified by *fp*. This function is equivalent to the fprintf function but receives "pointer to a variable argument list" as data to be output, not a variable argument list itself.

This function suffixes a null character to actual output character sequence however the null character is not counted as the number of output characters (return value). For more information on the format *control,* refer to the description of the fprintf function.

The *arg* indicating the variable argument list must be initialized by the va_start and va_arg macros. The vfprintf function does not call the va_end macro.

vprintf

Input/output function

Outputs a variable argument list to the standard output (stdout) by following the format.

Syntax

```
#include  < stdarg.h >
#include  < stdio.h >
int  vprintf ( const char *control,  va_list arg );
     control;        /* pointer to format string */
     arg;            /* pointer to variable argument list */
```

Return Value

Number of characters output	: Successful
Negative value	: Output error occurs

Description

The vprintf function converts and edits the variable arguments pointed to by *arg* according to the format *control* and then outputs the result to the standard output (stdout). This function is equivalent to the printf function but receives "pointer to a variable argument list" as data to be output, not a variable argument list itself.

This function suffixes a null character to actual output character sequence however the null character is not counted as the number of output characters (return value). For more information on the format *control*, refer to the description of the fprintf function.

The *arg* indicating the variable argument list must be initialized by the va_start and va_arg macros. The vprintf function does not call the va_end macro.

vsprintf

Input/output function

Outputs a variable arguments list to a memory area by following the format.

Syntax

```
#include  < stdarg.h >
#include  < stdio.h >
int  vsprintf ( char *s, const char *control, va_list arg );
     s;        /* pointer to space to which data is output */
     control;  /* pointer to format string */
     arg;      /* pointer to variable argument list */
```

Return Value

Number of characters output

Description

The vsprintf function converts and edits the variable arguments pointed to by *arg* according to the format *control* and then outputs the result to the area of memory pointed to by *s*. This function is equivalent to the sprintf function but receives "pointer to a variable argument list" as data to be output, not a variable argument list itself.

This function suffixes a null character to actual output character sequence however the null character is not counted as the number of output characters (return value). For more information on the format *control*, refer to the description of the fprintf function.

The *arg* indicating the variable argument list must be initialized by the va_start and va_arg macros. The vsprintf function does not call the va_end macro.

wcstombs

General utility function

Converts a wide string into a multibyte string.

Syntax

```
#include  < stdlib.h >
size_t  wcstombs ( char *s, const wchar_t *pwcs, size_t n );
     s;         /* multibyte string storage buffer */
     pwcs;      /* wide string */
     n;         /* number of bytes to be written */
```

Return Value

Number of characters converted	:	Successful (the number excludes the termination null)
-1	:	Invalid wide character is found during conversion (casting -1 to size_t upon returning)

Description

The maximum value of multibyte-characte is 3 byte. And even “euc”, ”sjis”, ”utf8”, in addition to “C” are supported as the locale.

The wcstombs function converts the number of bytes specified by *n,* out of the wide string specified by *pwcs,* into multibyte characters and stores the result into the buffer specified by *s*.

wctomb

General utility function

Converts a wide character into a multibyte character.

Syntax

```
#include  < stdlib.h >
int  wctomb ( char *s, wchar_t wchar );
     s;             /* multibyte character storage buffer */
     wchar;         /* wide character */
```

Return Value

- If *s* is NULL,

  0 : Current multibyte character set does not depend on the status.

  non-0 : Current multibyte character set depends on the status.

- If *s* is not NULL,

  number of bytes : The number of bytes of converted multibyte characters.

  0 : *wchar* is 0.

  -1 : A multibyte character correspond to *wchar* dose not exist.

Description

he maximum value of multibyte-characte is 3 byte. And even “euc”, ”sjis”, ”utf8”, in addition to “C” are supported as the locale.

The wctomb function converts the wide character specified by *wchar* into the corresponding multibyte character and stores the result into the buffer specified by *s*. When *s* is NULL, the function checks if the multibyte character set depends on the status at that time.

Chapter 10

The cc32R's Behavior

This chapter describes the C compiler(cc32R)'s behavior corresponding to “undefined behavior”, “implementation-defined behavior”, and “locale-specific behavior” in ANSI-C. Each description is preceded by its corresponding section number in the ANSI-C (American National Standard for Programming Languages - C, ANSI/ISO 9899-1990). The ANSI-C says as follows:

- Undefined behavior

  Behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which this International Standard imposes no requirements.

- Implementation-defined behavior

  Behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document.

- Locale-specific behavior

  Behavior that depends on local conventions of nationality, culture, and language that each implementation shall document.

10.1 Undefined Behavior

The operation dealt with as “undefined behavior” in ANSI-C is not guaranteed in the C compiler. In most cases, the result may be ignoring, the issuance of a diagnostic message, or occurrence of run-time error. Thus it is recommended to write a program that is free from “undefined behavior”.

Here follow operations corresponding to “undefined behavior” that may be probable (not guaranteed) in the C compiler cc32R. The number and the heading subsequent to “• ANSI-C” are the section number and the section heading of the corresponding ANSI-C, ANSI/ISO 9899-1990.

- ANSI-C 5.1.1.2 Translation Phases

  (End of source file)

  If no new-line character is present at the end of source file, a new-line character is automatically added (the last line of a file need not to end with a new-line character).

  If the source file ends with a new-line character preceded by a backslash, the backslash and the new-line character are deleted. If the source file ends at a midpoint of a preprocessing token Note 1) or of a comment, an error occurs.

  Note 1) Preprocessing token (see the ANSI-C 6.1)
  A minimal lexical element of text within a source file in C, and includes the following : header names, identifiers, preprocessing numbers, character constants, string literals, operators, punctuators, and single non-white-space character that do not match the above.

- ANSI-C 5.2.1 Character Sets

  (Characters other than those belonging to the character set)

  If characters that do not belong to the variable character set appear in a source file (excluding preprocess tokens not to be converted into tokens, character constants, string literals, header names, and comments), a warning is given, and the characters are ignored.

- ANSI-C 5.2.1.2 Multi-byte Characters

  If multi-byte characters are used anywhere other than comments, character-constant and string-literal the operation is not guaranteed. There can be instances in which the end of comment is not detected if what immediately precedes the end of comment, */, is in shift state.

- ANSI-C 6.1 Lexical elements

  (Pair of quotation marks)

  Either a ' or " that does not form a pair, if appears in a source, results in an error.

- ANSI-C 6.1.2.1 Scopes of identifiers

  Using the same identifier twice or more as a label within a function results in an error.

  Using an identifier that is not present in the current scope results in an error.

- ANSI-C 6.1.2 Identifiers

  Characters subsequent to the significant characters, if different in an identifier that identifies the same entity, result in a warning.

- ANSI-C 6.1.2.2 Linkages of identifiers

  If you declare the same identifier that stands for a function both for internal identifier and external identifier, it is regarded as an external identifier. If the identifier is put to static declaration in function definition, it is regarded as an internal identifier. If you define an identifier standing for anything else than functions for both internal identifier and external identifier, it is regarded as an internal identifier.

- ANSI-C 6.1.2.4 Storage durations of objects

  If a storage area reserved for an object that has automatic storage duration becomes no longer guaranteed, and if you use the pointer value that references the object, no error occurs during compilation but its operation is not guaranteed.

- ANSI-C 6.1.2.6 Compatible type and composite type

  If there are two declarations for the same object or for a function and if their types are not compatible, an error results.

- ANSI-C 6.1.3.4 Character constants

  A non-supported escape sequence, if appears either in a character constant or in a string literal, results in a warning, and the backslash is ignored (example: '\c' is interpreted as 'c').

- ANSI-C 6.1.4 String literals

Mixing wide character string literals and character literals

With regular string literal specified, if a wide character string literal token appears, the prefix character L is ignored and concatenated as a character string literal. With wide character literal specified, character strings cannot be concatenated.

- ANSI-C 6.1.7 Header names

The characters \,", or /*, if appear in a header name, are recognized as characters making up a file name (not processed as special characters).

- ANSI-C 6.2.1 Arithmetic operations

If the result of an arithmetic conversion cannot be expressed within a given space (deficiency in accuracy), an approximate value is used. In the case of conversion into an integer, however, digits to the right of the decimal points and the bit pattern of higher-order digits that couldn't be accommodated is discarded.

- ANSI-C 6.2.2.1 Lvalues and function designators

In other cases than initializing an array by use of an initialization expression, using an incomplete type on the lvalue results in an error.

- ANSI-C 6.2.2.2 void

Either accessing an object by use of a value of void type or adapting an implicit conversion to a void expression (except conversion into void) results in an error.

- ANSI-C 6.3 Expressions

Side effects

A side effect that occurs between sequence points of an expression is indeterminate. Do not write code that might lead to a different operation result due to a side effect.

For example, there can be chances in which `*p+5` of code “`*p++=*p+5`” is evaluated before or after `p++`, so that what is assigned `*p+5` turns indeterminate. In this instance, code a program in either way given below according to the purpose of process.

```
p=*p+5;
++p;
```

or

```
*(p+1)=*p+5;
p++;
```

Invalid operation, domain error

An invalid operation (division by 0 for example) results in an error. A domain error (an overflow, an underflow, or the like), if results from an operation, results in a warning. A warning is given only when either domain error is detected during compilation. In either case, the operation based on the arithmetic operation is not guaranteed.

- ANSI-C 6.3.2.2 Function calls

If the argument to a function is a void expression

Specifying a void expression other than null argument for an actual argument results in an error. If a null argument (void expression) is specified and if one or more formal arguments are defined for the function called, the value passed to the function is indeterminate.

Type incompatibility between argument and parameter

If the function is defined in a position where the function is not visible in calling a function with no function prototype declaration, and if the type of actual argument is not consistent with that of the formal argument after promotion (after executing implicit type conversion), the value of the actual argument is not guaranteed. For example, if an actual argument that has been declared as short (implicitly converted into int if no function prototype is given) is passed to a function that has a formal argument of unsigned int, an error occurs, but no error occurs if the said actual argument is passed to a function that has a formal argument of int.

Type incompatibility between function prototype and function definition

In calling a function when the function prototype declaration is visible, if the function is not defined for the type compatible with the declaration, an error results.

Prototype declaration of variable arguments

If a function that accepts a variable arguments list is called in a position where the function prototype that ends with "..." is not visible, there can be a chance that part of variable number of actual arguments is not correctly passed.

- ANSI-C 6.3.3.2 Address and indirection operators (Unary operators &, *)

  If you perform a reference as given below by use of an address arithmetic operator & or an indirect reference operator *, the operation is not guaranteed.

  - Referencing an invalid array
  - Referencing a null pointer
  - Referencing an object having automatic storage duration whose scope has expired

- ANSI-C 6.3.4 Cast operators

  If you cast a pointer toward a function to a pointer toward a function of different type, and if you call a function of a type incompatible with the original type, the operation is not guaranteed.

  You can cast a pointer toward a function to a pointer toward an object, and you can cast a pointer toward an object to a pointer toward a function.

  If you cast a pointer to an entity of non-integer type (non-character type either) or to an entity of non-pointer type, an error occurs in most cases. The operation of program is not guaranteed even though no error occurs.

- ANSI-C 6.3.6 Additive operation

  If you add/subtract a pointer toward an array and if the pointer points toward an area outside the area for the array elements, the value of the pointer is correctly derived (no error occurs). Referencing the content that a pointer points toward by use of the operator * allows you to reference the data stored there. The data is not an array element, so the operation of program is not guaranteed.

- ANSI-C 6.3.7 Bitwise shift operators

  If you specify a negative number for the extent of a shift operation or a number greater than the bit width of the expression to be shifted, the operation is not guaranteed. (An example of operation: if you give a negative shift extent, there can be instances in which the shift direction reverses. If the shift extent is greater than the bit width of the expression to be shifted, there can be a chance that the shift is correctly performed provided that the result can be expressed by the size that the type can afford.)

- ANSI-C 6.3.8 Relational operators (macro replacement)

  Even if a pointer to be compared by a relational operator (<, <=, >, >=) points neither to the same aggregate nor union, no error occurs; but the operation is not assured.

- ANSI-C 6.3.16.1 Simple assignment (simple assignment =)

  If you assign an object to objects in an overlapping manner, data in the overlapped part are not guaranteed.

- ANSI-C 6.5 Declarations

  If an object declared without linkage is incomplete even after the declaration ends (if the objects has an initial value) or even after the initial declaration ends, an error occurs.

- ANSI-C 6.5.1 Storage-Class Specifiers

  If a function is declared by use of a storage class specifier other than extern in a block scope, the operation is not guaranteed.

- ANSI-C 6.5.2.1 Structure and union specifiers

  Unnamed members

  If you define either a structure or a union made up of unnamed members only, the operation is not guaranteed.

  Type of bit field in structure

  Types valid for the declaration of a structure's bit field include char, short, int, and long, either signed or unsigned. If you declare any type other than these, the operation is not guaranteed.

- ANSI-C 6.5.3 Type qualifiers

  If you attempt to change an object declared as const by a value other than const on the left-hand side, that is, if you attempt to process an area declared as const by a cast or the like as if it is not const, the operation is not guaranteed (there can be instances in which no error occurs).

  If you attempt to change an object declared as volatile by a value other than volatile on the left-hand side, that is, if you attempt to process an area declared as volatile by a cast or the like as if it is not volatile, the operation is not guaranteed (there can be instances in which no error occurs).

- ANSI-C 6.5.7 Initialization

  If you use an object that has a non-initialized automatic storage duration before assigning a value to it, there can be instances in which a warning such as an error is not issued. It value is indeterminate.

  There can a chance that a warning such as an error is not issued in the instances given below, but the operation is not guaranteed.

  - An instance in which an object of either aggregate type or union type having a static storage duration has an initial value that are not enclosed in a pair of braces {   }.

  - An instance in which an object of either aggregate type or union type having an automatic storage duration has either an initialization expression of the type of the object or an initial value that are not enclosed in a pair of braces { }.

- ANSI-C 6.6.6.4 The return statement

  If a function's value referenced is not returned from the function, the function's value referenced is indeterminate.

- ANSI-C 6.7 External definitions

  If you define two or more identical identifiers having external linkage, and if they are included in one source, an error occurs during compilation; if they are separately included in two or more sources, an error occurs during linking.

- ANSI-C 6.7.1 Function definitions

  If the parameter list in defining a function that accepts variable arguments doesn't end with ", ...", and if an attempt is made to pass more arguments than the number declared in the parameter list, an error occurs.

- ANSI-C 6.7.2 External object definitions

  The identifier of an incomplete object having internal linkage is declared by an ambiguous definition, the operation is not guaranteed (there can be instances in which a warning is issued).

- ANSI-C 6.8.1 Conditional inclusion

  The token defined that is generated during expanding the preprocessing directive for #if or #elif is dealt with as an operator.

- ANSI-C 6.8.2 Source file inclusion

  If the preprocessing directive for #include agrees with neither of two header files, an error occurs.

- ANSI-C 6.8.3 Macro replacement

  A macro call on a function sequence having no arguments results in a error.

  A line that begins with # — a preprocessing directive, if present in the actual argument list in a macro call, is regarded as a preprocessing directive.

- ANSI-C 6.8.3.2 The # operator (character string formation)

  If string formation based on the operator # for preprocess doesn't result in a valid string constant, the operation is not guaranteed. There can be a chance that an error occurs during expansion.

- ANSI-C 6.8.3.3 The ## operator (token coupling)

  If coupling tokens with the operator ## for preprocess doesn't turn to a valid preprocess token, the operation is not guaranteed. For example, `func#1`, if expanded, turns to `func1`, but if `func1` is a meaningless token, then there can be a chance that an warning is issued during compilation or that an error occurs during linking.

- ANSI-C 6.8.4 Line control

  If the syntax of the preprocessing directive for #line after expansion is incorrect, an error occurs. In this instance, the line information is not updated.

- ANSI-C 6.8.8 Predefined macro names

  __LINE__, __FILE__, __DATE__, __TIME__, and __STDC__ are macros already defined. Defining them or canceling their definition by use of either #define or #undef causes an error.

- ANSI-C 7 Library

  If you attempt to copy an object to objects in an overlapping manner by use of a library function other than memmove, data in the overlapped part are not guaranteed.

- ANSI-C 7.1.2 Standard headers

  Including standard headers inside an external definition

  As for function declaration, object declaration, type definition, macro definition, and macro definition using a the same name as a keyword, the corresponding standard header files must have been included before the first reference. If you include them after referencing, they don't operate correctly.

  Redefining a reserved external name

  The process at the time of defining an external name other than program-reserved ones (an external name within a header, for example) depends on the linker.

- ANSI-C 7.1.4 Errors <errno.h>

  `errno` has been realized from macros and external variables. You can access `errno` even when you nullify the macro definition.

- ANSI-C 7.1.6 Common definition <stddef.h>

  Specifying a bit field member of a structure for the second parameter of the offsetof macro results in an error.

- ANSI-C 7.1.7 Use of library functions

  If the number of actual arguments of a library function is invalid, the operation of program is not guaranteed.

  If a library function that accepts variable arguments is not declared by means of including headers, there can be a chance that the function being handled doesn't operate correctly.

- ANSI-C 7.2 Diagnostics <assert.h>

  assert has been realized from macros. If you call assert by nullifying a macro call so as to access a function, a warning is issued during compilation, and an error occurs during linking due to the absence of external symbols.

- ANSI-C 7.3 Character handling <ctype.h>

  The argument to be passed to a character handling function is neither unsigned char nor EOF, the operation is not guaranteed.

- ANSI-C 7.6 Nonlocal jumps <setjmp.h>

  setjmp, if the macro definition is nullified, doesn't result in an error.

- ANSI-C 7.6.1.1 The setjmp macro

  Using the setjmp macro in the ways given below is recommended. If you use it in ways other than these, no error results, but if you use it in an complex expression, there can be a chance that part of the current execution environment (an interim result of expression evaluation, for example) is lost.

  - Controlling an operand in selection statement, iteration statement, and in comparing an integer constant expression (implicit process by use of the unary operator !, or the like).

  - Controlling an operand in a selection statement or an iteration statement

  - Expression statement (casting to void)

- ANSI-C 7.6.2.1 The longjmp function

  If you change an object of an automatic storage class that is not qualified as volatile during a period from executing setjmp to calling longjmp, the value of the object is not guaranteed.

  If the longjmp function starts up from a nested signal routine, it returns to the setjmp function, but the subsequent operation depends on the specifications of a signal function (a low-level function) you prepare.

- ANSI-C 7.7.1.1 The signal function

The signal function is not packaged in the C standard library. The process relating to a signal depends on the specifications of a signal function (a low-level function) you prepare. Thus the instances given below also depends on a signal function (a low-level function) you prepare.

- An instance in which a signal occurs as a result of calling the abort function or the raise function.
- An instance in which the signal handler calls a standard library function other than the signal function.
- An instance in which the signal handler references any static object of type other than volatile sig_atomic_t.
- An instance in which the value of errno is referenced after a signal occurs (except the result of calling the abort function or the raise function) and the corresponding signal handler calls the signal function that returns the value SIG_ERR.

- ANSI-C 7.8.1 Variable arguments list access macros

When a certain function (let this be A) invokes a function (let this be B) by using, as an actual argument, *ap* (pointer to variable argument list) updated by a va_arg macro, if the function B calls a va_arg macro by using *ap*, the following results.

- The function B (the function invoked by the function A) can carry out reference from variable arguments that *ap* indicates at the time when the function B is invoked.
- The function A (the function that invoked the function B) can carry out reference from variable arguments indicated by ap as it is at the time when the function A invokes the function B regardless of whether the function B references arguments of a variable number.

If you pass the *ap*'s address as an argument, or if you pass an aggregate (if *ap* is an aggregate) as an argument, ap of the function A at the time when returned from the function B continues from the value at the time when the function B terminates.

va_start, va_arg, and va_end have been realize from macros. If you invoke these by nullifying a macro call so as to access a function, a warning is issued during compilation, and an error occurs during linking due to the absence of external symbols.

- ANSI-C 7.8.1.1 The va_start macro

  If the second argument parmN of the va_start macro has been declared as a register class variable, function type, or array type, or if the declaration doesn't agree with the type as it becomes after the default actual argument promotions (the type resulting from applying implicit type conversion to an argument), the operation is not guaranteed.

- ANSI-C 7.8.1.2 The va_arg macro

  If the argument to be handled is not actually present in calling va_arg, the operation is not guaranteed.

  If the argument to be handled is not of the type specified in calling va_arg, the operation is not guaranteed.

- ANSI-C 7.8.1.3 The va_end macro

  Invoking va_end before invoking va_start doesn't cause an error, and the program operates properly.

  If a function having a variable argument list initialized by the va_start macro returns before the va_end macro is invoked, no error occurs; but the operation of program is not guaranteed.

- ANSI-C 7.9.5.2 The fflush function

  fflush for an input stream is ignored (no error returns).

- ANSI-C 7.9.5.3 The fopen function

  If neither the fflush function nor file positioning function[Note 2)] is invoked during a period from an input request to an output request for one stream, the input/output operation is not guaranteed.

Note 2) File positioning functions are : fgetpos, fseek, fsetpos, ftell, and rewind.

- ANSI-C 7.9.6 Formatted input/output functions

'printf'-like functions[Note 3)] / 'scanf'-like functions[Note 4)]

Though the type of function specifications doesn't agree with the corresponding number in the argument list, or the number of actual arguments are less than the number of conversion specifications, no error occurs, but the operation is not guaranteed. If the number of arguments is greater than specified by the conversion specifier, the excess arguments are ignored.

The input/output result for invalid conversion specifications in the 'printf'-like functions or the 'scanf'-like functions is indeterminate. In most cases, no error message is output. If input/output is different from what is expected, check that the code for conversion specifications is in correct format.

'printf'-like functions / 'scanf'-like functions %% conversion

In dealing with conversion specifications %% for the 'printf'-like functions or the 'scanf'-like functions, in most cases in which characters other than numeric characters are contained between % and %, the characters between % and % are subjected to input/output. For example, "%abcdef%" is converted into "abcdefg".

- ANSI-C 7.9.6.1 The fprintf function

Qualifier

In dealing with conversion specifications of the 'printf'-like functions, if the qualifier (the size specifying character, h or l) is specified previous to h or l preceding the conversion specifier other than the one involved (o, x, X, e, E, f, g, G), then the qualifier is ignored.

Flag

In dealing with conversion specifications of the 'printf'-like functions, if the flag # is specified previous to a conversion specifier other than the one involved (o, x, X, e, E, f, g, G), then the flag is ignored.

In dealing with conversion specifications of the 'printf'-like functions, if the flag 0 is specified previous to a conversion specifier other than involved (d, i, o, u, x, X, e, E, f, g, G), then the flag is ignored.

Note 3) The 'printf'-like functions : fprintf, printf, sprintf, vfprintf, vprintf, and vsprintf
Note 4) The 'scanf'-like functions : fscanf, scanf, and sscanf

Conversion result

Though an aggregate, a union, or a pointer toward an aggregate or a union is specified for an item other than %p and %s of the 'printf'-like functions, the program operates properly.

If the result of % conversion effected by the printf function exceeds 509 characters, the operation is not guaranteed.

- ANSI-C 7.9.6.2 The fscanf function

Qualifier

In dealing with conversion specifications of 'scanf'-like functions, if the qualifier (the size specifying character, h, l, L) is specified previous to a conversion specifier other than the one involved as given below, then the qualifier is ignored.

- Either h or l put previous to a conversion specifier other than d, i, n, o, u, or x.
- L put previous to a conversion specifier other than e, f, or g.

Compatibility with %p of 'printf'-like functions

The output format of the %p conversion of the 'printf'-like functions is compatible with the address format assigned to %p of the 'scanf'-like functions.

Storage area for the conversion result

If the area to which the resultant value of conversion effected by the 'scanf'-like functions is assigned is insufficient in capacity or of incompatible type, the operation is not guaranteed.

- ANSI-C 7.10.1 String conversion functions (conversion from a string into a numerical value)

If the conversion result effected by one of the functions (atof, atoi, atol) that converts a string into a numerical value cannot be expressed due to a domain error, the function returns the maximum value of the domain (HUGE_VAL, INT_MAX, or the like.).

- ANSI-C 7.10.3 Memory management functions (the free function, realloc function)

If you reference an area deallocated either by the free function or the realloc function, the operation is not guaranteed.

If you pass a value as given below to the first argument (the pointer toward an area to be deallocated) of the free function or the realloc function, the operation is not guaranteed.

· A value that is not a return value (the pointer toward an area whose allocation has been completed) of the calloc function, malloc function, or the realloc function

· A pointer toward an area that was formerly deallocated either by the free function or by the realloc function.

- ANSI-C 7.10.4.3 The exit Function

  If a program executes a call to the exit function twice or more (runs the atexit function so as to catalog the exit function, for example), the operation is not guaranteed.

- ANSI-C 7.10.6 Integer arithmetic functions

  If the result of running one of the arithmetic functions of integer type (abs, div, lab, ldiv) cannot be expressed, then the value cannot be guaranteed.

- ANSI-C 7.10.7 Multibyte character function (shift state)

  The shift state of the multi-byte exists. However, all functions implement processing in supposing that it is in absolutely the inital shift state when these functions were called.

- ANSI-C 7.11.2 Copying functions, 7.11.3 Concatenation functions

  The operation of these functions is not guaranteed in such instances as given below.

  If the size of the copy destination is less than the copy source in running the memcpy, memmove, strcpy, or strncpy function.

  If the area for storing the result of concatenating character strings is insufficient in running either the strcat function or the strncat function.

- ANSI-C 7.12.3.5 The strftime function

  If a character that is not a conversion specifier is present in the time conversion format *format* of the strftime function, then the character is output without being changed.

A subtraction involved in two pointers that don't point toward the same array doesn't result in an error, provided that all the types of operands (pointers) are compatible. But if you make the types compatible by casting, the operation result is not guaranteed.

A mismatch between the number of arguments and that of parameters

If the number of actual arguments does not agree with that of formal parameter in calling a function at a position where the function's prototype is invisible, the values of unmatched dummy arguments are not guaranteed.

Concatenating wide string literals

With wide string literals specified, concatenated strings turn invalid.

Accessing an object by use of an lvalue

If you assign to an object a value on an lvalue having an incompatible type, the operation is not guaranteed.

10.2 Implementation-defined Behavior

Here follows how the operation dealt with as “implementation-defined behavior” in ANSI-C goes on in the C compiler cc32R, such as how messages are notified, the number of significant characters for an identifier, or the format of integer or floating-point number, etc.

The number and the heading subsequent to “• ANSI-C” are the section number and the section heading of the corresponding ANSI-C, ANSI/ISO 9899-1900. Each issue as “implementation-defined behavior” is shown in a pair of angular brackets < >, and the corresponding behavior of cc32R is detailed subsequent to < >.

10.2.1 Translation

- ANSI-C 5.1.1.3 Diagnostics

<How a diagnostic is identified.>
The diagnosis messages include information messages, warning messages, error messages, and fatal error messages. For details of output formats of messages, see Chapter 12 "Messages from the C Compiler".

10.2.2 Environment

- ANSI-C 5.1.2.2.1 Program startup

<The semantics of the arguments to main.>
The arguments passed to the main function depend on the specifications of the startup program you prepare.

- ANSI-C 5.1.2.3 Program execution

<What constitutes an interactive device.>
The operation of input/output devices depends on the specifications of the read function and the write function (low-level functions) you prepare.

10.2.3 Identifiers

- ANSI-C 6.1.2 Identifiers

<The number of significant initial character (beyond 31) in an identifier without external linkage.>
As for an identifier without external linkage, the first 240 characters are significant. The 241st character and subsequent ones are ignored.

<The number of significant initial character (beyond 6) in an identifier with external linkage.>
As for an identifier, the first 240 characters are significant. The 241st character and subsequent ones are ignored. Identifiers are case-sensitive.

<Whether case distinctions are significant in an identifier with external linkage.>
You can describe only an integer constant expression in the case statement.

10.2.4 Characters

- ANSI-C 5.2.1 Character sets

<The members of the source and execution character sets, except as explicitly specified in this International Standard.>
Both the source character set and execution character set comply with JIS X 0201,0208 character set. (The Latin character part of JIS X 0201 will be regarded as ASCII.)
The EUC(Expanded Unix Code),Shift-JIS and UCS-2(UTF-8 encoded) are supported as the actual character-encoding.

- ANSI-C 5.2.1.2 Multibyte characters

<The shift states used for the encoding of multibyte characters.>
The shift state (character strings that indicate the beginning and the end of a multibyte character) for multibyte characters is not available.

- ANSI-C 5.2.4.2.1 Sizes of integral types <limits.h>

<The number of bits in a character in the execution character set.>
8 bits.

- ANSI-C 6.1.3.4 String literals

<The mapping of member of the source character set (in character constants and string literals) to members of the execution character set.>
If the M32RKIN enviromen variable is not “utf8”, the arrangement correspondence between the characters in the source character set and those in the execution charcter set is one to one. Otherwise, the characters that have same meaning and different character-code may be changed to normalize with the character set of JIS X 0201 and JIS X 0208.

<The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant .>
If the M32RKIN enviroment variable is not “utf8” ,the most right edge of 2 byte in this character will be combined by the big-endian order.
Otherwise,this character will be changed to 0xffff.

<The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character.>
If this character constant is not wide-character, it is the value of the rightmost characters in a string literal.
The wide-character constatnt is processed in accordance with the M32RKOUT enviroment variable.

<The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant.>
The “euc”,”sjis” and “utf8” locales are supported.

- ANSI-C 6.2.1.1 Characters and integers

<Whether a "plain" char has the same range of values as signed char or unsigned char.>
"char" behaves like "signed char" on the code CC32R generated.Yet, the CC32R processes "char" with regarding as different model from "signed char" when a interpreting.

10.2.5 Integers

- ANSI-C 6.1.2.5 Types

<The representations and sets of values of the various types of integers.>
For internal representation and the limit values of various integer data, see 5.2 “Integral Types”. The C compiler interprets int as equivalent to signed int, short to signed short, and long to signed long. Yet, the C compiler interprets assuming that char and signed char differ when the grammar interpretation. But the generated binary code executes as equivalent to char and signed char.

- ANSI-C 6.2.1.2 Signed and unsigned integers

<The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented.>
In converting an integer into a "signed integer less in size than the original", the lower-order bits of the original integer are converted into a signed integer without being changed. The most significant bit of the singed integer after conversion is used for the sign bit.

In converting an unsigned integer into a "signed integer of the same size", the lower-order bits are converted into a signed integer without being changed.

- ANSI-C 6.3 Expressions

<The results of bitwise operations on signed integers.>
The bit operation of a signed integer is dealt with as that of unsigned integer.

- ANSI-C 6.3.5 Multiplicative operators

<The sign of the remainder on integer division.>
The sign of the remainder is the same as that of the dividend.

- ANSI-C 6.3.7 Bitwise shift operators

<The result of a right shift of a negative-valued signed integral type.>
A right shift of negative-valued signed integer is an arithmetic right shift.

10.2.6 Floating-Point

- ANSI-C 6.1.2.5 Types

<The representations and sets of values of the various types of floating-point numbers.>
Refer to 5.3 “Floating Types” for internal representations and sets of values of floating types.

- ANSI-C 6.2.1.3 Floating and integral

<The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value.>
Numbers are rounded to the nearest value within representable range of the floating type that is the result of conversion.

- ANSI-C 6.2.1.4 Floating types

<The direction of truncation or rounding when floating-point number is converted to a narrower floating-point number.>
Numbers are rounded to the nearest value within representable range of the floating type that is the result of conversion.

10.2.7 Arrays and Pointers

- ANSI-C 6.3.3.4 The sizeof operator, 7.1.1 Definitions of terms

<The type of integer required to hold the maximum size of an array - that is, the type of the sizeof operator, size_t .>
The type of the sizeof operator,size_t,is defined as unsigned long.

- ANSI-C 6.3.4 Cast operators

<The result of casting a pointer to an integer or vice versa.>
In converting a pointer to an integer or vice versa, all the bits are used with the notation unchanged, so a correct conversion

results.

- ANSI-C 6.3.6 Additive operators, 7.1.1 Definitions of terms

<The type of integer require to hold the difference between two pointers to elements of the same array, ptrdiff_t .>
The type of integer that holds the difference between two pointers, ptrdiff_t, is defined as int.

10.2.8 Registers

- ANSI-C 6.5.1 Storage-class specifiers

<The extent to which objects can actually be placed in registers by use of the register storage-class specifier.>
The register storage-class specifier is ignored.

10.2.9 Structures, Unions, Enumerations, and Bit-fields

- ANSI-C 6.3.2.3 Structure and union members

<A member of a union object is accessed using a member of a different type.>
The bit pattern stored in the member of union is accessed, and the value is interpreted according to the type of the member accessed.

- ANSI-C 6.5.2.1 Structure and union specifiers

<The padding and alignment of members of structures. This should present no problem unless binary data written by one implementation are read by another.>
Details of padding and alignment of bit field, see 5.9 "Bit Fields".

<Whether a "plain" int bit-field is treated as a signed int bit-field or as an unsigned int bit-field.>
A normal bit field of int type is dealt with as a bit field of signed int type.

<The order of allocation of bit-fields within a unit.>
Bit fields are allocated from high-order to low-order in storage.

<Whether a bit-field can straddle a storage-unit boundary.>
A bit field is not allocated across an alignment boundary.

- ANSI-C 6.5.2.2 Enumeration specifiers

<The integer type chosen to represent the values of an enumeration type.>
An enumeration type is treated as a int.

10.2.10 Qualifiers

- ANSI-C 6.5.3 Type qualifiers

<What constitutes an access to an object that has volatile-qualified type.>
Each reference to a volatile object name will constitute one access to the object. volatile-qualified objects are not optimized.

10.2.11 Declarators

- ANSI-C 6.5.4 Declarators

<The maximum number of declarators that may modify an arithmetic, structure, or union type.>
No limit is placed on the maximum number of declarators.

10.2.12 Statements

- ANSI-C 6.6.4.2 The switch statement

<The maximum number of case values in a switch statement.>
The maximum value of case in a switch statement depends on the available memory capacity.

10.2.13 Preprocessing Directive

- ANSI-C 6.8.1 Conditional inclusion

<Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matched the value of the same character constant in the

execution character set. Whether such a character constant may have a negative value.>
The value of a single-character constant in the constant expression that controls conditional inclusion agrees with the value of the same character constant in the execution character set. Such character constant can be assigned a negative value.

- ANSI-C 6.8.2 Source file inclusion

<The method for locating includable source files.>
The sequence of retrieving header files specified by #include is given in the Function column for the -I option (-I path) in Table 3.6 in 3.2 "Startup Options of the C Compiler".

<The support of quoted names for includable source files.>
In the #include preprocess instruction, you can specify a file name to include by means of enclosing it in a pair of quotation marks (" ").

<The mapping of source file character sequences.>
The values of ASCII characters are respectively assigned to the characters in the source file.

- ANSI-C 6.8.6 Pragma directive

<The behavior on each recognized #pragma directive.>
Unrecognized #pragma directives are ignored.

- ANSI-C 6.8.8 Predefined macro names

<The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available.>
__DATE__and __TIME__are always available.

10.2.14 Library Functions

- ANSI-C 7.1.6 Common definitions <stddef.h>

<The null pointer constant to which the macro NULL expands.>
NULL (the null pointer) is defined as ((void *) 0).

- ANSI-C 7.2 Diagnostics <assert.h>

<The diagnostic printed by and the termination behavior of the assert function .>

The diagnostic message from assert is :

`Assertion failed:`*expression*`, file:`*file_name*`, line:`*line_number*

- ANSI-C 7.3.1 Character testing functions

<The set of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint, and isupper functions.>

Characters (ASCII characters) for which isalnum, isalpha, iscntrl, islower, isprint, or isupper returns true are as follows.

Function	Tested character set
isalnum	'0'-'9', 'A'-'Z', 'a'-'z'
isalpha	'A'-'Z', 'a'-'z'
iscntrl	0x00-0x1F, 0x7F
islower	'a'-'z'
isupper	'A'-'Z'
isprint	0x20-0x7E

- ANSI-C 7.5.1 Treatment of error conditions

<The value returned by the mathematics functions on domain errors.>

If a definition area error occurs in a numerical value calculating function, EDOM is set to `errno`.

<Whether the mathematics functions set the integer expression errno to the value of the macro ERANGE on underflow range errors.>

If an underflow occurs in a numerical value calculating function, ERANGE is set in `errno`.

- ANSI-C 7.5.6.4 The fmod function

<Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero.>

If the second argument of the fmod function is 0, a domain error occurs, and EDOM is set in errno (the calculation result is not guaranteed).

- ANSI-C 7.7.1.1 The signal function

<The set of signals for the signal function.>

It is depends on the user-defined signal function (a low-level function).

<The semantics for each signal recognized by the signal function.>

It is depends on the user-defined signal function (a low-level function).

<The default handling and the handling at program startup for each signal recognized by the signal function.>

They are depend on the user-defined signal function (a low-level function).

<If the equivalent of signal (sig, SIG_DEL); is not executed prior to the call of a signal handler, the blocking of the signal that is performed.>

It is depends on the user-defined signal function (a low-level function).

<Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function.>

It is depends on the user-defined signal function (a low-level function).

- ANSI-C 7.9.2 Streams

<Whether the last line of a text stream requires a terminating new-line character.>

Whether the line feed character is required depends on the specifications of the read function and the write function (low-level functions) you prepare. Functions in the C standard library used for calling them are so designed that they operate properly though the line feed code is missing from the last line.

<Whether space characters that are written out to a text stream immediately before a new-line character appear when read in.>

Whether the null character is output depends on the specifications of the read function and the write function (low-level functions) you prepare.

<The number of null characters that may be appended to data written to a binary stream.>

No null characters are appended to a binary stream.

- ANSI-C 7.9.3 Files

<Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file.>
The file position indicator is initially located at the end of file.

<Whether a write on a text stream causes the associated file to be truncated beyond that point.>
Truncating the associated file is not caused.

<The characteristics of file buffering.>
Three methods are available for buffering — full buffering, line buffering, and no buffering. You can set/change them by use of the setbuf function or the setvbuf function.

<Whether a zero-length file actually exists.>
It is depends on user-defined low-level functions.

<The rules for composing valid file names.>
They depend on user-defined low-level functions.

<Whether the same file can be open multiple times.>
It is depends on the user-defined open function (a low-level function).

- ANSI-C 7.9.4.1 The remove function

<The effect of the remove function on an open file.>
It is depends on the user-defined remove function (a low-level function).

- ANSI-C 7.9.4.2 The rename function

<The effect if a file with the new name exists prior to a call to the rename function .>
It is depends on the user-defined rename function (a low-level function).

- ANSI-C 7.9.6.1 The fprintf function

<The output for %p conversion in the fprintf function.>

The output for %p in the 'printf'-like functions[Note 5)] turns equal to %X.

- ANSI-C 7.9.6.2 The fscanf function

<The input for %p conversion in the fscanf function.>

The input for %p in the 'scanf'-like functions[Note 6)] turns equal to %X.

<The interpretation of a - character that is neither the first nor the last character in the scanlist for %[conversion in the fscanf function.>

The conversion %[in the 'scanf'-like functions allows to indicate an inclusive range of characters by using '-' (hyphen) as follows, assuming that the character set is ASCII :

%[	**Character set**
%[a-f]	'a'-'f' (i.e., 'a', 'b', 'c', 'd', 'e', and 'f')
%[0-9][a-f]	'0'-'9' and 'a'-'f'
%[--a]	'-' through 'a'
%[---]	'-' only
%[z-a]	Nothing
%[-af]	'-', 'a', and 'f ' ('-' at the top do not indicate a range)
%[af-]	'a', 'f', and '-' ('-' at the end do not indicate a range)
%[a-f-z]	'a'-'f', '-', and 'z'

- ANSI-C 7.9.9.1 The fgetpos function, 7.9.9.4 The ftell function

<The value to which the macro errno is set by the fgetpos or ftell function on failure.>

`errno` is set to EFSEEK if the result of the fgetpos or ftell function is unsuccessful (i.e., an error was returned from the environment).

Note 5) The 'printf'-like functions : fprintf, pointf, sprintf, vfprintf, vprintf, and vsprintf

Note 6) The 'scanf'-like functions : fscanf, scanf, and sscanf

- ANSI-C 7.9.10.4 The perror function

<The messages generated by the perror function.>
It is depends on the user-defined _strerror function (a low-level function).

- ANSI-C 7.10.3 Memory management functions

<The behavior of the calloc, malloc, or realloc function if the size requested is zero.>
NULL is returned.

- ANSI-C 7.10.4.1 The abort function

<The behavior of the abort function with regard to open and temporary files.>
It is depends on the user-defined abort function (a low-level function).

- ANSI-C 7.10.4.3 The exit function

<The status returned by the exit function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE.>
It is depends on the user-defined _exit function (a low-level function).

- ANSI-C 7.10.4.4 The getenv function

<The set of environment names and the method for altering the environment list used by the getenv function.>
It is depends on the user-defined getenv function (a low-level function).

- ANSI-C 7.10.4.5 The system function

<The contents and mode of execution of the string by the system function.>
It is depends on the user-defined system function (a low-level function).

- ANSI-C 7.11.6.2 The strerror function

<The contents of the error message strings returned by the strerror function.>
It is depends on the user-defined _strerror function (a low-level function).

- ANSI-C 7.12.1 Components of time

<The local time zone and Daylight Saving Time.>

The environment variable TZ is used to set the local time (the calender time in each locale). As the local time, JST (default), EST5EDT, CST6CDT, MST7MDT, PST8PDT, and UTC are supported. As daylight saving time, EST5EDT, CST6CDT, and MST7MDT are supported.

- ANSI-C 7.12.2.1 The clock function

<The era for the clock function.>

It is depends on the user-defined clock function (a low-level function).

10.3 Locale-specific Behavior

Here follows how the operation dealt with as “locale-specific behavior” in ANSI-C goes on in the C compiler cc32R.

The number and the heading subsequent to “• ANSI-C” are the section number and the section heading of the corresponding ANSI-C, ANSI/ISO 9899-1900. Each issue as “locale-specific behavior” is shown in a pair of angular brackets < >, and the corresponding behavior of cc32R is detailed subsequent to < >.

- ANSI-C 5.2.1 Character sets

<The content of the execution character set, in addition to the required members.>
The character code set of JIS X 0201 (except for the Latin characters) and JIS X 0208 were extended.

- ANSI-C 5.2.2 Character display semantics

<The direction of printing.>
Always left to right.

- ANSI-C 7.1.1 Definitions of terms

<The decimal-point character.>
The decimal point is 0x2E (‘.’) in the all locales.

- ANSI-C 7.3 Character handling <ctype.h>

<The implementation-defined aspects of character testing and case mapping functions.>
Same as “• ANSI-C 7.3.1 Character testing functions” in 10.2 “Implementation-defined Behavior”, 10.2.14 “Library Functions”.

The behavior of macros declared and functions defined in ctype.h in the all locales are the same as in “C” locale.

- ANSI-C 7.11.4.4 The strncmp function

<The collation sequence of the execution character set.>
In the all locales, the collation sequence of the character set under the strncmp function is the same as that of ASCII.

- ANSI-C 7.12.3.5 The strftime function

<The formats for time and date.>

The strftime function allows to represent time and date in the all enviroments(locales) as follows :

- Date *mm1/dd/yy*
  (*mm1* is the month, *dd* is the day, and *yy* is low- order 2 digits of the year in Christian Era.)
- Time *hh* : *mm2* : *ss*
  (*hh* is hours, *mm2* is minutes, and *ss* is seconds.)

Chapter 11

Low-level Library

11.1 The Low-level Library Programming

11.1.1 The Low-level Library for the C Standard Library

Part of actions of the library functions, such as standard input/output, memory management, signal handling, time manipulation, etc. depend on the target system. In the C standard library, the functions dependent on the target system are separated as low-level functions (low-level library), and input/output specifications of the process functions (low-level functions, see Table 11.1 and Table 11.2) that achieve respective actions are defined.

In the C standard library, the actions dependent on the target system are effected by means of calling the low-level functions. To use actions dependent on the target system in a C program, you must prepare necessary low-level functions.

Specifications of individual low-level functions are given in 11.2 “The Low-level Functions Specifications”. Table 11.3 shows which C standard library function uses which low-level function.

Table 11.1 Low-level Functions (Used by C Standard Library)

Low-level Function	Description
open	Opens a file
close	Closes a file
read	Reads data from a file
write	Writes data into a file
lseek	Reposition the read/write point in a file
_get_core	Allocates a memory area
_rel_core	Frees a memory area
getuniqnum	Obtains the unique number for each process
_strerror	Gets the error message corresponding to an error number
_exit	Exits a program

Table 11.2 shows the functions which are defined as low-level functions in C standard library but defined as entry functions in ANSI. For specifications for these functions, refer to Chapter 9 "C Standard Library".

Table 11.2 Functions defined as low-level functions in cc32R

Header	Function	Description
signal.h	raise	Send a signal to the executing program.
	signal	Sets up a signal handler that responds to the signal.
stdio.h	remove	Deletes a file.
	rename	Renames a file.
stdlib.h	getenv	Gets the content of an environmental variable.
	system	Passes a command string to the host environment.
time.h	clock	Gets the elapsed processor time.
	time	Reads the current calendar time.

Table 11.3 shows C standard library functions which need low-level library listed in Table 11.1 and Table 11.2.

Table 11.3 C Standard Library vs Low-level Library (1/3)

Header	C Standard Library Function	Required Low-level Function(s)
assert.h	assert	_get_core, _rel_core, write, lseek, raise
locale.h	localeconv	_get_core, _rel_core
	setlocale	_get_core, _rel_core
signal.h	raise	raise
	signal	signal
stdio.h	perror	write, _get_core, _rel_core, lseek, _strerror
	fclose	write, getuniqnum, _rel_core, close, remove, lseek
	fflush	write, lseek
	fopen	open, close, lseek
	freopen	write, getuniqnum, _rel_core, close, open, remove, lseek
	remove	remove
	rename	rename
	tmpfile	getuniqnum, open, close, lseek
	tmpnam	getuniqnum
	fgetpos	lseek

Table 11.3 C Standard Library vs Low-level Library (2/3)

Header	C Standard Library Function	Required Low-level Function(s)
stdio.h (Cont.)	fseek	lseek, write
	fsetpos	lseek, write
	ftell	lseek
	rewind	lseek, write
	fgetc	read, _get_core, _rel_core
	fgets	read, _get_core, _rel_core
	fputc	write, _get_core, lseek, _rel_core
	fputs	write, _get_core, lseek, _rel_core
	getc	read, _get_core, _rel_core
	getchar	read, _get_core, _rel_core
	gets	read, _get_core, _rel_core
	putc	write, _get_core, _rel_core, lseek
	putchar	write, _get_core, _rel_core, lseek
	puts	write, _get_core, _rel_core, lseek
	ungetc	_get_core, rel_core
	fread	read, _get_core, _rel_core
	fwrite	write, _get_core, _rel_core, lseek
	fprintf	write, _get_core, _rel_core, lseek
	fscanf	read, _get_core, _rel_core
	printf	write, _get_core, _rel_core, lseek
	scanf	read, _get_core, _rel_core
	sprintf	write, _get_core, _rel_core, lseek
	sscanf	read, _get_core, _rel_core
	setbuf	_rel_core
	setvbuf	_rel_core
	vfprintf	write, _get_core, _rel_core, lseek
	vprintf	write, _get_core, _rel_core, lseek
	vsprintf	write, _get_core, _rel_core, lseek

Table 11.3 C Standard Library vs Low-level Library (3/3)

Header	C Standard Library Function	Required Low-level Function(s)
stdlib.h	abort()	raise()
	calloc()	_get_core(), _rel_core()
	free()	_rel_core()
	getenv()	getenv()
	malloc()	_get_core(), _rel_core()
	realloc()	_rel_core(), _get_core()
	system()	system()
time.h	asctime()	write(), _get_core(), _rel_core(), lseek()
	clock()	clock()
	ctime()	write(), _get_core(), getenv(), _rel_core(), lseek()
	localtime()	getenv()
	strftime()	write(), _get_core(), getenv(), _rel_core(), lseek()
	time()	time()

||||| Note |||||

Before starting C program, initial setting of the low-level libraries is required. To do so, write the initial setting program in the startup program.

11.1.2 Input/Output with the Low-level Library

Files of standard input and output functions of C standard library level are controlled with file type data. Files of low-level library are controlled with "file numbers". A file number is an integer corresponding to the file number of the actual file and starts with 0.

File numbers 0, 1 and 2 should be assigned to a standard input, standard output and standard error output, respectively. Typically, 0 is input from console, 1 and 2 are output to console. For file numbers 0 to 2, prepare low-level libraries which are compatible with execution environment for file number. When 0 to 2 are already assigned, make sure that the open function will not return values 0, 1 and 2.

Low-level library's open function assigns the file number to the given file path name (the name to identify the file in the file system). The open function must set the following information so that the file can be input and output by using this file number.

- Device type of file (console, printer, disk file, etc.) :

  Give unique name to files of special devices such as console and printer and identify these files with the open function.

- Information on buffer location and size :

  This information is necessary when buffering files.

- Byte offset from the start of a file to the next reading or writing location :

  The byte offset is necessary to read/write disk file.

By using the information set by the open function, read/write the file (read/write function), set read/write location (lseek function). The close function reads contents of file buffer and writes the contents into the file, allowing the data area set by the open function to be used again.

11.2 The Low-level Functions Specifications

This section describes specifications of functions for low-level library in alphabetical order. The specifications include **Syntax** (interface to call the function), **Summary** (general description), **Return Value** (*value* : *meaning*) and **Description** (operation, precaution, example, etc.).

_exit

Low-level function

Syntax

```
void _exit( int ex_num );
```

ex_num; /* exit status code of program */

Summary

Exits a program.

Return Value

None

Description

The _exit function terminates the program run and returns the exit status of the program to the environment.

This function signals the environment that the program is to end. Write the end process of the current environment in this function definition.

_get_core

Low-level function

Syntax

`void *_get_core ( int size );`

size ;   /* the size of data to be allocated */

Summary

Allocates a memory area.

Return Value

The start address of the allocated area : Successful
(void*) 0 : Failed

Description

The _get_core function allocates a memory area. *size* is the size of memory area to be allocated. If *size* is smaller than the available space to allocate, an error is occurred.

When allocation is successful, the start address of the location is returned. Otherwise, (void*)0. Refer to examples shown below :

```
#define  HEAPSIZE bytes ------  Specify the size (bytes) of the area managed by _get_core().
static union{
    int dummy;
    char heap [HEAPSIZE];
}heap_area;              ------  Declare the data area for allocation.  This is union which
                                 has the int type member as the first menber to align in
the
                                 4-byte boundary.

static char *brk = heap_area.heap;    ------ Initialize by substituting the start
                                             address to be allocated.

void *_get_core(int size)
{
    char *p;

    if( brk + size > heap_area.heap + HEAPSIZE) --- Check the area for
                                                    remainder.
        return((void *)0);
    p = brk;
    brk += size;          ---------- Update the end address of the allocated area.
    return((void *)p);    ---------- Return the start address of the allocated area.
}
```

_rel_core

Low-level function

Syntax

```
void _rel_core ( void *ptr );
```

ptr ; /* pointer to the memory space to be freed */

Summary

Frees a memory area.

Return Value

None

Description

This function frees a memory area being managed by the _get_core function is deallocated. *ptr* is the pointer to the memory area to be released.

_strerror

Low-level function

Syntax

```
char *_strerror( int error_number );
```

error_number ; /* user-defined error number */

Summary

Gets the error message corresponding to an error number.

Return Value

Pointer to an error message (character string)

Description

The _strerror function returns the pointer to the error message corresponding to the error number *error_number*.

close

Low-level function

Syntax

`int close ( int` *file_no* `);`

file_no ;   /* file number of the file to be closed */

Summary

Closes a file.

Return Value

0	: Successful
-1	: Failed

Description

The close function closes the file denoted to by the file number *file_no* which is gotten by the open function.

This function clears the file control information storage area set by the open function so that the area can be used again. When file buffering is used within low-level library, this function outputs the buffer content to the actual file.

This function returns 0 when the file is successfully closed, if not, -1.

getuniqnum

Low-level function

Syntax

```
int getuniqnum (void);
```

Summary

Obtains the unique number for each process.

Return Value

The value specific to a unit of the program operations.
This value must be kept unchanged during program is running.

Description

The getuniqnum function returns the number by which the environment specifies the operation unit when two or more programs are to run simultaneously. The number may differ from operation unit to unit even though these units have the same execution code. However, it must be the same for the same operation unit codes. This function returns the same value to environment where the program performs only one operation.

lseek

Low-level function

Syntax

```
long lseek (int file_no, long offset, int base );
```

file_no ;	/* target file number */
offset ;	/* offset (bytes) indicating the read/write point */
base ;	/* origin of offset */

Summary

Reposition the read/write point in a file.

Return Value

The new offset (bytes) from the read/write position to in a file	: Successful
-1	: Failed

Description

The lseek function sets the read/write position as measured in bytes in the file denoted by *file_no*. The new position is set by the third parameter (*base*) as shown below :

(a) *base* is 0	At the position, beginning of the file plus *offset* bytes.
(b) *base* is 1	At the position, current position plus *offset* bytes.
(c) *base* is 2	At the position, file size plus *offset* bytes.

When the file is interactive device, for example, console, printer or the like, if the new offset value is negative or setting of (a) or (b) exceeds the file size, error occurs. When the correct file position is set, this function returns the new read/write position represented by the *offset* from the beginning of the file. Otherwise, it returns -1.

open

Low-level function

Syntax

`int open (const char *`*name*`, int `*mode*`, int `*flag*` );`

name ;	/* string denoting the file path name */
mode ;	/* specification of mode upon opening the file */
flag ;	/* specification of process upon opening the file (always 0666) */

Summary

Opens a file.

Return Value

File Number of the open file	: Successful
-1	: Failed

Description

The open function prepares process to handle the file which corresponds to the path name of the file specified by *name*. This function must determine the type of file (console, printer, disk or file) for later reading/writing. The type of file will be referenced every time read/write process is taken by using the file number returned from the open function. The second parameter *mode* specifies the process which must be taken upon opening the file. Each bit of the data has the following function.

	0	23	24	25	26	27	28	29	30	31
mode	......	(g)		(f)	(e)	(d)			(c)	(a,b)

Bit	Description
(a) (31st bit)	When 0, indicates that the file is read only. When 1, indicates that the file is write only.
(b) (31st bit)	Indicates that the file is read and write when both this bit and the 31st bit are 0. Note that this bit is not set to 1 when the 31th bit is 1.
(c) (30th bit)	When 1, the read/write point of the next file can be set at the end of the file; and when 0, at the beginning of the file.

(*cont.*)

Bit	Description
(d) (27th bit)	When 1, indicates that a new file can be opened if the file indicated by path name is not found.
(e) (26th bit)	When1, and if the file indicated by the path name is found, indicates that the contents of the file are discarded and file size is set to 0.
(f) (25th bit)	When 1, indicates that a file is opened on binary mode. When 0, indicates that a file is opened on text mode.
(g) (23rd bit)	When 0, the file is opened in text mode. If the bit is 1, the file is opened in binary mode.

If the file process specified by mode is not compatible with the nature of the real file, an error is occurred. When the file is successfully open, this function returns the number (integer, 0 or larger) of the file which will be used in read, write, lseek and close routines. The file number and corresponding real file must be controlled by low level library. -1 is returned if the file cannot be opened.

read

Low-level function

Syntax

```
int read (int file_no, char *buffer, unsigned int count );
```

file_no ;	/* file number of the file to be read */
buffer ;	/* pointer to the area in which the data read is stored */
count ;	/* number of bytes to be read */

Summary

Reads data from a file.

Return Value

Number of bytes actually read	: Successful
-1	: Failed

Description

This function reads a range of data within the bytes indicated by count from the file indicated by file number file_no and stores it in the area indicated by buffer. The read/write position in the file is advanced forward by the number of bytes read. When the function was able to read normally, return the number of bytes read; if it failed to read, return the value "-1."

If there was no data to read and the end of file (EOF) was reached, return the value 0, and not -1.

write

Low-level function

Syntax

`int write (int` *file_no*`, const char *`*buffer*`, unsigned int` *count* `);`

file_no ;	/* file number of the write target file */
buffer ;	/* pointer to the area into which data is written */
count ;	/* bytes of data to be written */

Summary

Writes data into a file.

Return Value

Number of bytes actually written	: Successful
-1	: Failed

Description

The write function writes the *count*-byte data stored in the area pointed to by *buffer* into the file denoted by *file_no*. The read/write point in the file advances by the bytes written. When writing is successful, the number of bytes actually written is returned. Otherwise, -1 is returned.

Chapter 12

Single-precision Mathematical Function Library

This library is a version of the C standard mathematic function library (defined in the math.h header) that has been turned into single-precision arithmetic. It helps to increase the efficiency of applications using conventional double-precision mathematic function libraries (by increasing the execution speed and reducing the code size).Some single-precision mathematic functions use the FPU instructions and others do not.

12.1 Composition of Functions

Table 12.1 lists the functions in the single-precision mathematic function library. Check the C language formats of prototype declarations shown in the table to find the types of return values and arguments of these single-precision mathematic functions.

Table 12.1. Single-precision Mathematic Function Library

Single-precision mathematic functions (C language formats of prototype declarations)		Function	Corresponding math.h functions
Those do not use the FPU instructions	Those that use the FPU instructions		
float cosf(float)	float cosf5(float)	Cosine	cos
float sinf(float)	float sinf5(float)	Sine	sin
float tanf(float)	float tanf5(float)	Tangent	tan
float acosf(float)	float acosf5(float)	Inverse cosine	acos
float asinf(float)	float asinf5(float)	Inverse sine	asin
float atanf(float)	float atanf5(float)	Inverse tangent	atan
float atan2f(float,float)	float atan2f5(float,float)	Inverse tangent (with division)	atan
float coshf(float)	float coshf5(float)	Hyperbolic cosine	cosh
float sinhf(float)	float sinhf5(float)	Hyperbolic sine	sinh
float tanhf(float)	float tanhf5(float)	Hyperbolic tangent	tanh
float powf(float,float)	float powf5(float,float)	Power	pow
float sqrtf(float)	float sqrtf5(float)	Positive square root	sqrt
float ceilf(float)	float ceilf5(float)	Integral value with fractions rounded up	ceil
float expf(float)	float expf5(float)	Exponential function	exp
float fabsf(float)	float fabsf5(float)	Absolute value	fabs
float floorf(float)	float floorf5(float)	Integral value with fractions rounded down	floor
float fmodf(float,float)	float fmodf5(float,float)	Remainder	fmod
float frexpf(float,int*)	float frexpf5(float,int*)	Resolve into 0.5-1.0 values and power of 2	rexp
float ldexpf(float,int)	float ldexpf5(float,int)	Multiply power of 2	ldexp
float logf(float)	float logf5(float)	Natural logarithm	log
float log10f(float)	float log10f5(float)	Base-ten logarithm	log10
float modff(float,float*)	float modff5(float,float*)	Resolve into integral and fractional parts	modf

Specifications for these single-precision mathematic functions have been determined based on the C standard double-precision mathematic functions according to the rules described below.

[1] Operation

The basic operations of these functions are the same as each double-precision mathematic functions from which they have been derived, except that internal arithmetics are performed with the type float.

[2] Function name

The functions that do not use the FPU instructions are named after the double-precision mathematic functions from which they have been derived, by adding the letter 'f' to each. The functions that use the FPU instructions are named in the same way by adding the string 'f5' to each.

[3] Type of argument and return value

The arguments and return values have their types changed from double to float.

12.2 Using the Library

12.2.1 Header File

Before the single-precision mathematic functions can be used, one of the following header files must be included. Choose either one that suits to your need.

mathf.h	When you use only the single-precision mathematic functions.
math.h	When you use the C standard double-precision mathematic functions or the single-precision mathematic functions.

Note that because the header math.h includes the functions of mathf.h, you do not need to include mathf.h if you already have math.h included. The following shows the contents written in these header files.

[1] Prototype declaration of functions

The prototype declaration of single-precision mathematic functions are written in mathf.h, while the prototype declaration of both double-precision and single-precision mathematic functions are written in math.h.

[2] Function name replacement

In mathf.h, if the -m32re5 option (to use the FPU instructions of the M32R-FPU core) is specified when compiling the source file, calls to the single-precision mathematic functions that do not use the FPU instructions are changed to calls to the equivalent functions in the single-precision mathematic function library that use the FPU instructions.

```
Example 1: When -m32re5 is specified at the same time
        #include <mathf.h>   /* <math.h> also acceptable */
        ans = cosf(rd);
                ↓ Compiled with -m32re5 added
        ans = cosf5(rd);  /* Becomes equivalent to this */
```

This replacement for the cosf function case, for example, is accomplished by a macro like the one shown below.

```
#define cosf cosf5
         :
(Defined the same way for other single-precision
mathematic functions)
         :
```

[Caution]

The function name that is stored in the load module when a called function name is replaced, is the replaced function name and not the pre-replacement function name. Therefore, pre-replacement function names can be neither specified nor displayed in the debugger (e.g., M3T-PD32R) or TM inspector.

(You can specify or display replaced function names, though.)

12.2.2 Link with the Single-precision Mathematic Function Library

The single-precision mathematic function library is contained in the C standard library (e.g., m32RcR.lib) included with your package. Link the C standard library in the same way as you would use conventional C standard libraries.

12.3 Precautions

12.3.1 Dynamic range

In single-precision representation (type float), the magnitude of representable values is smaller than in double-precision (type double). When you replace the double-precision mathematic functions with single-precision mathematic functions, make sure the input or output values will not exceed the range of values representable by single-precision numbers.

12.3.2 About error handling

The error occurring conditions listed below are handled the same way as for the C standard double-precision mathematic functions. Because these conditions vary with each function, refer to the Section 9.3, "Details of C Standard Library Functions" in the user's manual to confirm function specifications for the corresponding double-precision mathematic functions.

- Conditions in which a domain error (EDOM) occurs
- Conditions in which a range error (ERNAGE) occurs
- Conditions in which an overflow value (HUGE_VAL) is returned to the calling function
- Conditions in which an underflow value (0) is returned to the calling function

Chapter 13

The set of 64-bit integer arithmetic functions

The set of functions to perform C language integer arithmetic in the 64-bit dynamic range has been added to the standard library. As for the integer type in C language, these functions can perform the four fundamental operations in arithmetic, as well as bitwise, shift and compare operations in the 64-bit range.

13.1 Header file long64.h

For the 64-bit integer arithmetic functions to be used, the header file long64.h must be included.
In long64.h, the necessary types, constants and function prototypes are declared.

(1) Type name
The type (structure) holds a 64-bit integer. All of the 64-bit integer arithmetic functions use this type as they input and output 64-bit integers.

Signed 64-bit integer... .... LONG64
Unsigned 64-bit integer... ...ULONG64

(2) Constant
The constant represents the maximum and minimum values of a 64-bit integer.

LONG64_MAX... ... ...Maximum value of LONG64
LONG64_MIN... ... ... Minimum value of LONG64
ULONG64_MAX... ... Maximum value of ULONG64

(3) Prototype declaration
Prototype for a 64-bit integer arithmetic function is declared.

13.2 Function structure

Comprised mainly of the C language operators that are put into functions.

(1) Arithmetic operation functions
Performs a 64-bit integer arithmetic operation, with the result returned by 64-bit integer type.

* Four rules of arithmetic (addition, subtraction, multiplication and division), remainder and monadic negative

(2) Bitwise operation functions

Performs a 64-bit integer bitwise operation, with the result returned by 64-bit integer type.

* Bitwise shifts (left shift, right logical shift, right arithmetic shift)
* Bitwise logical operations (OR, AND, exclusive-OR and inversion)

(3) Comparison and determination functions

Compares a 64-bit integer or determines if it is 0, with the result returned by int type.

* Comparison
* Determination of whether or not 0

(4) Type conversion functions

Converts type from 64-bit integer to C language integer or floating-point type or vice versa.

*	Signed 64-bit integer	<---->	unsigned 64-bit integer
*	Signed 64-bit integer	<---->	float or double
*	Unsigned 64-bit integer	<---->	float or double
*	Signed 64-bit integer	<---->	long or unsigned long
*	Unsigned 64-bit integer	<---->	long or unsigned long

(5) Other functions

Sets immediate data or replaces a decimal string, etc. with a 64-bit integer.

* Immediate set
* String to 64-bit integer conversion

These 64-bit integer arithmetic library functions are listed in Table 13.1 (1) to (5) below.

For the return values and return types of these functions, refer to the C language prototype declaration formats shown in the table.

Table 13.1 64-bit integer arithmetic functions (1) Arithmetic operation functions(1/2)

function name (Prototype declaration of C language)	Perform (S)=signed (U)=unsigned	Operation
[Four rules of arithmetic, remainder and monadic negative]		
LONG64 addl64(LONG64 n1, LONG64 n2);	(S) addition	n1 + n2
LONG64 subl64(LONG64 n1, LONG64 n2);	(S)subtraction	n1 - n2
LONG64 mull64(LONG64 n1, LONG64 n2);	(S)multiplication	n1 * n2
LONG64 divl64(LONG64 n1, LONG64 n2);	(S)division	n1 / n2
LONG64 modl64(LONG64 n1, LONG64 n2);	(S)remainder	n1 % n2
LONG64 negl64(LONG64 n1);	(S)monadic negative	-n1
ULONG64 addul64(ULONG64 n1, ULONG64 n2);	(U)addition	n1 + n2

Table 13.1 64-bit integer arithmetic functions (1) Arithmetic operation functions(2/2)

function name (Prototype declaration of C language)	Perform (S)=signed (U)=unsigned	Operation
ULONG64 subul64(ULONG64 n1, ULONG64 n2);	(U)subtraction	n1 - n2
ULONG64 mulul64(ULONG64 n1, ULONG64 n2);	(U)multiplication	n1 * n2
ULONG64 divul64(ULONG64 n1, ULONG64 n2);	(U)division	n1 / n2
ULONG64 modul64(ULONG64 n1, ULONG64 n2);	(U) remainder	n1 % n2
ULONG64 negul64(ULONG64 n1);	(U) monadic negative	-n1

Table 13.1 64-bit integer arithmetic functions (2) Bitwise operation functions

function name (Prototype declaration of C language)	Perform (S)=signed (U)=unsigned	Operation
[Bitwise shifts (left shift, right logical shift, right arithmetic shift)]		
LONG64 shltl64(LONG64 n1, unsigned int nbit);	(S)left shift	n1 << nbit
LONG64 shrtl64(LONG64 n1, unsigned int nbit);	(S)right arithmetic shift	n1 >> nbit
ULONG64 shltul64(ULONG64 n1,unsigned int nbit);	(U)left shift	n1 << nbit
ULONG64 shrtul64(ULONG64 n1,unsigned int nbit);	(U)right arithmetic shift	n1 >> nbit
[Bitwise logical operations (OR, AND, exclusive-OR and inversion)]		
LONG64 orl64(LONG64 n1, LONG64 n2);	(S)OR	n1 ｜ n2
LONG64 andl64(LONG64 n1, LONG64 n2);	(S)AND	n1 & n2
LONG64 xorl64(LONG64 n1, LONG64 n2);	(S)exclusive-OR	n1 ^ n2
LONG64 notl64(LONG64 n1);	(S)inversion	~n1
ULONG64 orul64(ULONG64 n1, ULONG64 n2);	(U)OR	n1 ｜ n2
ULONG64 andul64(ULONG64 n1, ULONG64 n2);	(U)AND	n1 & n2
ULONG64 xorul64(ULONG64 n1, ULONG64 n2);	(U)exclusive-OR	n1 ^ n2
ULONG64 notul64(ULONG64 n1);	(U)inversion	**~n1**

Table 13.1 64-bit integer arithmetic functions (3) Comparison and determination functions

function name (Prototype declaration of C language)	Perform (S)=signed (U)=unsigned	Operation (Returns)
[Comparison]		
int cmpl64(LONG64 n1, LONG64 n2);	(S)Comparison	n1 > n2 --> positive n1 == n2 --> 0 n1 < n2 --> negative
int cmpul64(ULONG64 n1, ULONG64 n2);	(U)Comparison	
[Determination of whether or not 0]		
int evall64(LONG64 n1);	(S)Determination	n1 == 0 --> 0 n1 != 0 --> not 0
1int evalul64(ULONG64 n1);	(U)Determination	

Table 13.1 64-bit integer arithmetic functions (4) Type conversion functions

function name (Prototype declaration of C language)	Conversion
[Signed 64-bit integer <----> unsigned 64-bit integer]	
ULONG64 l64_to_ul64(LONG64 input);	LONG64 --> ULONG64
LONG64 ul64_to_l64(ULONG64 input);	ULONG64 --> LONG64
[Signed 64-bit integer <----> float or double]	
float l64_to_float(LONG64 input);	LONG64 --> float
double l64_to_double(LONG64 input);	LONG64 --> double
LONG64 float_to_l64(float input);	float --> LONG64
LONG64 double_to_l64(double input);	double --> LONG64
[Unsigned 64-bit integer <----> float or double]	
float ul64_to_float(ULONG64 input);	ULONG64 --> float
double ul64_to_double(ULONG64 input);	ULONG64 --> double
ULONG64 float_to_ul64(float input);	float --> ULONG64
ULONG64 double_to_ul64(double input);	double --> ULONG64
[Signed 64-bit integer <----> long or unsigned long]	
long l64_to_long(LONG64 input);	LONG64 --> long
unsigned long l64_to_ulong(LONG64 input);	LONG64 --> unsigned long
LONG64 long_to_l64(long input);	long --> LONG64
LONG64 ulong_to_l64(unsigned long input);	unsigned long --> LONG64
[Unsigned 64-bit integer <----> long or unsigned long]	
long ul64_to_long(ULONG64 input);	ULONG64 --> long
unsigned long ul64_to_ulong(ULONG64 input);	ULONG64 --> unsigned long
ULONG64 long_to_ul64(long input);	long --> ULONG64
ULONG64 ulong_to_ul64(unsigned long input);	unsigned long --> ULONG64

Table 13.1 64-bit integer arithmetic functions (5) Other functions

function name (Prototype declaration of C language)	Perform (S)=signed (U)=unsigned	Operation
[Immediate set]		
LONG64 imml64(signed long shigh, unsigned long ulow);	(S)Immediate set	shigh<<32 + ulow
ULONG64 immul64(unsigned long uhigh, unsigned long ulow);	(U)Immediate set	uhigh<<32 + ulow
[String to 64-bit integer conversion]		
LONG64 strtol64(const char *s, char **endptr, int base);	(S)String->Integer	64-bit version of strtol
ULONG64 strtoul64(const char *s, char **endptr, int base);	(U)String->Integer	64-bit version of strtoul

13.3 Method for using the functions and example usage

To use the functions, create a program as described below.

(1) Include long64.h.

(2) Define the variable in which to store a value with either LONG64 or ULONG64.

(3) From Table 8, select the function corresponding to the relevant type for the necessary operation and call it in your program.

(4) Hold the return value in a variable of type LONG64 or ULONG64.

The set of 64-bit integer arithmetic functions are included in the C standard library (e.g., m32RcR.lib) that is included with your software. Link the C standard library in the same way as when using the conventional C standard library functions.

The following shows a function as a programming example that performs “a = (b * (c - 10)) >> 33” with signed 64 bits and returns the result after being converted to long type.

```
#include <long64.h>
LONG64 a, b, c;
long
func(void)
{
    LONG64 s1, s2, s3;
    long k;
    s1 = long_to_l64(10L);     /* s1 = (LONG64)10L */
    s2 = subl64(c,s1);         /* s2 = c - s1 */
    s3 = mull64(b,s2);         /* s3 = b * s2 */
    a = shrtl64(s3,33);        /* a = s3 >> 33 */
    k = l64_to_long(a);        /* k = (long)a */
    return k;

}
```

13.4 Notes

13.4.1 Precautions regarding the sign

In the 64-bit arithmetic functions, no operations can be performed where signed and unsigned types coexist. When using the arithmetic functions, please make sure that all of the 64-bit integers input and output unanimously are either signed or unsigned.

Example:

When adding a signed (LONG64) variable 'b' to an unsigned (ULONG64) variable 'a' and assigning the sum to an unsigned variable 'c'

```
Incorrect:  c = add164(a,b); or c = addu164(a,b);

    b is converted to unsigned type, and all are changed to
    unsigned type

Correct:    c = addu164(a, 164_to_u164(b));
```

Chapter 14

Messages from the C Compiler

14.1 Getting Execution Result of the C Compiler

You can judge the execution result yielded by the C compiler by looking into the messages and the exit status.

14.1.1 Message Format

Upon encountering an error condition, the C compiler outputs the error message describing the error status to the standard error output, in the following format :

- Syntax

  input_information : *message_type* : *message*

  Note) "*input_information* :" is output only when necessary.

- Pattern

  file name : *message_type* : *message*
  file name , line number : *message_type* : *message*
  <command line> : *message_type* : *message*
  message_type : *message*

  Note: Underlined items are *input_information* (no the underline is output).

- Example

```
"smp.c", line 2: error: unterminated #ifdef conditional
```

File name Line number Message type Message

```
<command line>: error: macro name missing after -D option
```

Message type Message

```
"a.c", line 9: warning: main: function has no return statement
```

File name Line number Message type Message

14.1.2 Message Types

Messages are classified into three types depending on their severity, as shown in Table 14.1

Table 14.1 Message Type

Message Type	When an Error Occurs
Information	Outputs an information message and continues processing.
Warning	Outputs a warning message and continues processing.
Command line error	Outputs an error message which corresponds to the command line and stops processing.
Error	Outputs an error message and stops processing.
Fatal error	Outputs an error message and stops processing.

For details of messages, see 14.2 “Message Lists”.

14.1.3 Exit Status

Upon completion of the execution, the C compiler returns the exit status (value showing the execution result) as shown in Table 14.2.

Table 14.2 Exit Status

Exit Status	Result
0	Complete successfully or warning occurs.
1	Error occurs.

14.2 Message Lists

Tables 14.2 to 14.7 show messages of the C compiler alphabetically. 'xxx' in the messages means arbitrary input information: an input file name, and an identifier, a name, numbers, strings in a source cord.

14.2.1 Information Messages

Table 14.3 Information Messages

Message	Description
`constant out of range due to unportable conversion`	A constant will be out of the range of the type in other implementations.
`less complete than prior compatible declaration — this declaration ignored`	This declaration contains less declaration elements than in the previous compatible declaration. This declaration is ignored.
`more complete than prior compatible declaration — prior declaration ignored`	This declaration contains more declaration elements than in the previous compatible declaration. The prior declaration is ignored.
`prior identical declaration — ignored`	There is an identical declaration before this. This declaration is ignored.
`some garbage after` #*xxx* `argument`	There is an improper character after #*xxx*.
`unnecessarily punctuated parameter list in #define`	The parameter list in a function-like macro definition is incomplete under #define.
#*xxx* `with no argument`	Argument(s) is(are) missing from #*xxx*.

14.2.2 Warning Messages

Table 14.4 Warning Messages (1/4)

Message	Description
argument type inconsistent with (imputed) declaration	The type of an argument is inconsistent with the declaration.
array subscript imply one element initialized by 0	A subscript of an array contains an element that has been initialized to 0.
calling a non-function object: “*xxx*”.	The source code contains an attempt to call the object *xxx* that is not a function.
‘*ch*’: illegal escape sequence	An escape sequence is improper.
xxx: conflicts with prior option – ignored	The option *xxx* conflicts with prior option. The prior option takes effect.
constant out of range due to unportable conversion	A constant may fall outside the range of the type when porting.
xxx: empty macro definition; regarded as empty	The -D option lacked the macro definition. (No definition is given after -D *xxx*=.)
EOF in character constant – constant truncated	EOF is contained in a character constant. The constant is truncated.
EOF or NUL in string literal – string truncated	Either EOF or NUL is contained in a character string literal. The character string is truncated.
fewer arguments specified than in (imputed) definition	There is shortage of arguments designated.
floating-point number overflow	A floating-point number has overflowed.
floating-point number underflow	A floating-point number has underflowed.

Table 14.4 Warning Messages (2/4)

Message	Description
function called before declared	A function is called before declared.
function does not return value with defined return type	A function has not returned a return value in the type defined.
function has no return statement	The return statement is missing from a function.
illegal character in input file — ignored illegal character	A invalid character is present in the input file. The invalid character is ignored.
xxx is referenced before set	The name *xxx* was referenced without being assigned to.
length of character constant is greater than 512 — constant truncated	The length of a character constant has exceeded 512 characters. The constant has been truncated.
length of identifier name is 240 characters — name truncated	The length of an identifier is up to 240 characters. The name has been truncated.
length of identifier name is 31 characters — name truncated	The length of an identifier is up to 31 characters. The name has been truncated.
length of string literal is greater than 512 — string truncated	The length of a character string literal has exceeded 512 characters. The character string has been truncated.
xxx: may be referenced before set	There is a chance that the name *xxx* may have been referenced without being assigned.
missing terminal ‘’’’ for string — assumed at end of line	The termination of a character string “ is missing. The end of line has been assumed to be termination.
mixed normal and wide characters in the same string — concatenated string omitted	Regular characters and wide characters are mixed in a single character string. The concatenated string has been omitted.

Table 14.4 Warning Messages (3/4)

Message	Description
`more arguments specified than in (imputed) definition`	The number of arguments designated is too many.
xxx`: multiple optimization options — ignored`	The command line option for optimization is duplicated. The option *xxx* is ignored.
`nothing declared in current declaration — ignored`	No declaration has been made in a declaration line. The declaration has been ignored.
`number of parameters not equal between use and definition`	The number of arguments doesn't agree with that given in the prototype declaration.
`parameter incompatible with previous use`	There is no compatibility between arguments.
`#`*xxx*`: requires exactly 1 identifier`	What is to be specified by #*xxx* is a single identifier.
`shift count greater than number of bits`	The count of shift has turned greater than the number of bits.
`storage specifier conflicts with prior declaration — this declaration ignored`	The storage class designation doesn't agree with that previously declared. This declaration has been ignored.
`the characters /* are found in a comment`	The characters /* is found inside a comment.
`too many digits in floating-point number; extra digits ignored`	The number of digits of a floating-point number is too many. The excess digits have been ignored.
xxx`: unable to optimize — skipped phase`	Optimization of the input file *xxx* is impossible. The optimization steps have been passed by.
xxx`: undefined name in #if constant expression; regarded as 0`	An undefined name *xxx* is present in the constant expression given after #if. It is regarded as 0.

Table 14.4 Warning Messages (4/4)

Message	Description
`unknown commandline option — ignored`	This command option is unsupported. The option is ignored.
`unknown directive — directive ignored`	This directive is unsupported. The directive is ignored.
`unknown size static global array used`	You cannot specify this size for a static global array.
xxx`: unknown suffix, passed to linker`	The extension of the input file *xxx* cannot be determined (neither .c, .ms, nor .mo). It is regarded as an object file, and its name is left unchanged and passed to the linker.
`unportable character constant`	The value of a character constant is troublesome in portability.
`unrecognized #pragma — directive ignored`	This is not a #pragma directive supported. The directive is ignored.
`\x is hex escape sequence`	A character that represents a hexadecimal number has not been specified after \x (\x is the prefix of hexadecimal number code).

14.2.3 Command Line Error Messages

Table 14.5 Command Line Error Messages

Message	Description
`xxx: can’t execute`	The subordinate process *xxx* cannot be started up. Check the setting of the environment variable M32RLIB or files in the default directory, and set the environment so that the subordinate process can be started up.
`xxx: invalid multicharacter option`	The option *xxx* does not recognize.
`xxx: invalid parameter`	The parameter specified after the option *xxx* is improper. Example: In -R old=new, neither of P, D, C, and B has been specified for old; or nothing has been specified for new.
`xxx: invalid unicharacter option`	The option *xxx* is not available.
`xxx: missing parameter`	A necessary argument is not specified after the option *xxx*.
`-: there is no option`	An option is not specified.

14.2.4 Error Messages

Table 14.6 Error Messages (1/15)

Message	Description
#*xxx* after #else	#*xxx* is present after #else.
aggregate/union type object must have constant expression initializer list	You must initialize an aggregate and a shared set by use of a constant expression.
argument name not specified in function header	No argument is specified for a function header.
#*xxx* argument starts with a digit	The first argument of the preprocessing directive #*xxx* starts with a numeric character (it must be an identifier).
arguments given to macro '*xxx*'	Arguments are required on the macro *xxx*.
array subscript requires combination of pointer to object type and integral type	An array must be a combination of pointers and entities of integer type.
at most one storage class specifier per declaration	Two or more storage class specifiers are present in a declaration (you can specify only a single storage class specifier in a declaration).
'auto' must appear within a function	You must declare an auto variable within a function.
badly punctuated parameter list in #define	An improper character is present in an argument of #define.
bit field members must have integral types	A member of bit field must be of integer type.
bit field size must be integral constant expression	A bit field size must be specified by an integer constant.
bit field size must not be negative	A bit field size must be specified by a positive number.

Table 14.6 Error Messages (2/15)

Message	Description
`bit field size too large`	A size specified for a bit field is too large.
`block scope initialization not allowed for external declaration`	You must not initialize an external variable within a block.
`break statement in invalid context`	You cannot use break here.
`call to non-function attempted`	You have attempted to perform a function call to an entity that is not a function.
`cannot const a function type object`	You cannot make a const declaration on an entity of function type.
`cannot initialize a typedef variable`	You cannot perform initialization when making a typedef declaration.
`cannot start or end with ## operator`	The ## operator must not lie at the beginning or the end of a replacement character string list.
`cannot typedef function definition`	You cannot make typedef declaration on a function definition.
`case label must be an integral constant expression`	You must specify an integer constant expression for case.
`case or default label in invalid context`	You can use neither case nor default here.
`cast type must be “void” or scalar type`	You must specify either void or an entity of scalar type for cast.
`character constant must fit its storage range`	The value of a character constant must be within a proper range.
`constant expression evaluated out of range`	The value of a constant expression has exceeded a proper range.

Table 14.6 Error Messages (3/15)

Message	Description
constant should fit its storage range	The value of a constant must be within a proper range.
continue statement in invalid context	You cannot use continue here.
controlling expression of if statement must have scalar type	An expression within an if statement must be of scalar type.
controlling expression of iteration statement must have scalar type	An expression within an iteration statement must be of scalar type.
controlling expression of switch statement must have integral type	An expression within a switch statement must be of integer type.
xxx: '-D option' must be followed by an identifier	Specify an identifier for the -D option.
'defined' is followed by invalid macro name	The macro name after defined is improper (it must be an identifier).
xxx: 'defined' is followed by invalid macro name	The macro name after defined is improper (it must be an identifier).
dereference a pointer to void	You cannot perform an indirect reference to a pointer toward void.
directive has unexpected non-whitespace preceding newline	An improper character is present in the #line declaration.
division by zero	A division by zero is contained.
division by zero in #if constant expression	An attempt has been made to perform division by 0 in an expression under #if.
double quoted strings not allowed in #if constant expression	You cannot specify a character string enclosed in a pair of quotation marks in an expression under #if.

Table 14.6 Error Messages (4/15)

Message	Description
`duplicate case label in switch statement`	case is duplicated in a switch statement.
`duplicate definition of enumeration-constant`	A definition is duplicated in an enum constant.
`empty constant expression in #if`	An expression is missing from #if.
`enum used as type is incomplete or undefined`	Either an incomplete or an undefined enum is used.
`enumeration-constant out of range`	The value of enum constant has exceeded a proper range.
`EOF in comment`	The end of comment is not present.
`expected parameter list missing -or- missing type for variable`	Either an argument or an initializer is not found. Or the type of a variable is indefinite.
`fewer arguments are specified than declared`	The number of arguments is less than declared.
`first operand of conditional operator must have scalar type`	The first term of a conditional operation must be of scalar type. A ternary operation must be of scalar type.
`first operand of “.” is not struct/union type`	The operator . (period) must be used for a structure or a union.
`first operand of “->” must be pointer to struct/union type`	The operator -> must be used for a pointer of a structure or a union.
`floating-point numbers are not allowed in #if constant expression`	You cannot specify a floating-point number in an expression under #if.
`function already defined`	The function has already been defined.

Table 14.6 Error Messages (5/15)

Message	Description
function cannot be an array element	You cannot make a function an array element.
function cannot return a function	A function cannot return an entity of function type.
function cannot return an array	A function cannot return an array.
function missing parameter declaration	No arguments have been declared on a function.
function return type incompatible with previous declaration	The return-type of the function is different from that declared earlier.
function return type is not declared	The return-type of the function has not been declared.
garbage after end of constant expression in #if	An improper character is present after a constant expression under #if.
identifier already used as member name in this struct/union	The member name has already been used.
identifier cannot have type “void”	You cannot define an identifier as an entity of void type.
identifier is not member of left hand side struct/union	You used a member which is not the declared member of a structure or union.
xxx: identifier is required between ‘(‘ and ‘)’ in `defined ( )’	You must put an identifier in ( ) of defined.
identifier must be defined as a typedef-name	You used a name which is not the typedef-defined name.
identifier not member of left-hand side struct/union	You used a member which is not the declared member of a structure or union.
identifier redeclared in current declaration	An identifier declared in a duplicate manner is present.

Table 14.6 Error Messages (6/15)

Message	Description
identifier redefined in current declaration	An identifier defined in a duplicate manner is present.
identifier undeclared in current declaration	The specified identifier has not been declared.
identifier undefined	An undefined identifier is present.
identifier with no linkage and incomplete object type	There is an identifier of incomplete type without linkage.
illegal combination of types in initialization	The combination of types in performing initialization is incorrect.
illegal floating-point constant	The description of a floating-point number is incorrect.
illegal hexadecimal constant	The description of a hexadecimal digit is incorrect.
illegal octal constant	The description of an octal digit is incorrect.
implicit declaration conflicts with prior (possibly implicit) declaration	An implicit declaration is inconsistent with a previous implicit declaration.
initializer must be constant expression	Initialization must be in terms of a constant expression.
integer character constant requires one or more multibyte characters enclosed in single-quotes	A character constant has be so formed that one or more characters are enclosed in a pair of single quotation marks.
invalid character constant in #if	The way of specifying a character constant in #if is erroneous.
invalid combination of types in assignment	The combination of types in an assignment is incorrect.

Table 14.6 Error Messages (7/15)

Message	Description
invalid file name in #*xxx*	A file name specified in the preprocessing directive #*xxx* is incorrect.
xxx: invalid file name in #include	A file name specified in the preprocessing directive #include is incorrect.
invalid initializers	Initialization has been performed in an incorrect manner.
invalid line number in #line	The way of specifying a line number in #line is improper.
xxx: invalid macro name	A macro name is incorrect.
invalid macro name	A macro name is incorrect.
invalid parameter name in #define	A parameter name in #define is improper.
invalid token in #if constant expression	An improper token is present in a constant expression under #if.
#*xxx* is not within a conditional	The position of the preprocessing directive #*xxx* is improper.
‘)’ is required after ‘defined (macroname’	The right parenthesis for enclosing an identifier after defined is missing (the identifier is not enclosed).
left operand of assignment operator must be modifiable lvalue	The lvalue of an assignment expression must be modifiable.
less parameters than definition	There is a shortage of the number of parameters in a function prototype declaration.
line number shall not specify zero, nor a number greater than 32767	A line number must be between 1 and 32767.

Table 14.6 Error Messages (8/15)

Message	Description
xxx: macro body/parameter-list redefined	The source code attempted to redefine either a macro or an argument list.
macro name '*xxx*' is reserved	*xxx* is a predefined macro.
macro name missing after -D option	A macro name hasn't been specified for the -D option.
macro name missing after -U option	There is no macro name after the -U option.
member fields with bit size of zero must not be named	You cannot give a name to a bit field whose size is 0.
more arguments are specified than declared	The number of arguments is greater than declared.
more parameters than definition	The number of parameters in a function prototype declaration is too many. The number of parameters does not agree with that in a function definition.
new style function can not have an old style parameter declaration	An old way of declaring arguments must not be mixed with a new way.
newline character not allowed	You cannot include a line feed character in a character constant.
no arguments to macro '*xxx*'	No arguments are present in the macro *xxx*.
no repetition of type qualifier in declaration	You cannot declare a type qualifier repeatedly.
xxx: No such file or directory	The file to be included *xxx* cannot be read.
non-integral constant-expression for enumeration-constant	An attempt has been made to define a non-integral constant for an enum constant.

Table 14.6 Error Messages (9/15)

Message	Description
`nonintegral initialization of bitfield`	An attempt has been made to initialize a bit field with an entity of non-integer.
`#: not first non-whitespace character`	# has been written in a position other than the beginning of a line.
`not in any file`	_FILE_ cannot be expanded.
`object with block scope and external or internal linkage can not be initialized`	You cannot initialize an object with external/internal linkage.
only *xxxa* argument(s) to macro '*xxx*b' (*xxx*c argument(s) expected)	There is a shortage of the number of arguments in a macro (the macro *xxxb* requires *xxxc* arguments but no more than *xxxa* arguments have been specified).
`only leftmost dimension of array may be incomplete`	The leftmost dimension of an array is incomplete.
`only one default statement allowed per switch statement`	You can describe no more than one default statement in a switch statement.
`operand of cast can not be type "void"`	The operand of a cast to void must have scalar type.
`operand of equality operator has invalid type`	The type of operand of the equivalence operator is incorrect.
`operand of "-" has invalid type`	The type of operand of the - operator is incorrect.
`operand of "+" has invalid type`	The type of operand of the + operator is incorrect.
`operand of "++"/"--" must be modifiable lvalue`	The lvalue of the ++ operator and –– operator may be modifiable.
`operand of "++"/"--" must have arithmetic type or pointer to object type`	The operand of the ++ operator and –– operator must have arithmetic or pointer type.

Table 14.6 Error Messages (10/15)

Message	Description
operand of “~” must have integral type	The operand of the ~ operator must be integer.
operand of “++”/”--” must have scalar type	The operand of the ++ operator and –– operator must have scalar type.
operand of “!” must have scalar type	The operand of the ! operator must have scalar type.
operand of “sizeof” must not be bitfield	You may not specify a bit field for the operand of the sizeof operator.
operand of “sizeof” must not be function type	You may not specify an function type for the operand of the sizeof operator.
operand of “sizeof” must not be incomplete type	You may not specify an incomplete type for the operand of the sizeof operator.
operand of unary “&” must be lvalue or function designator	The operand of the & address operator must be either an lvalue or a function specifier.
operand of unary “*” must be pointer type	The operand of the * indirect reference operator must be pointer type.
operand of unary “+”/”-” must have arithmetic type	The operand of the + unary operator and - unary operator must be arithmetic type.
operand of unary “&” must not refer to bitfield	You cannot use the & address operator for a bit field.
operand of unary “&” must not refer to register object	You cannot use the & address operator for an object brought under a register declaration.
operands of “*”/”/” (multiply/divide) must have arithmetic type	The operand of the * (multiplication) operator and / (division) operator must be arithmetic type.
operands of “&”/”^”/”|” must have integral type	The operand of the & operator, ^ operator, and | operator must be arithmetic type.

Table 14.6 Error Messages (11/15)

Message **Description**

`operands of “<<“/”>>” must have integral type`

The operator of the << operator and >> operator must be integral type.

`operands of “%” must have integral type`

The operand of the % operator must be integral type.

`operands of “&&”/”||” must have scalar type`

The operand of the && operator and || operator must be scalar type.

`operands of relational operator have invalid type`

The type of the relational operator (<,<=,>,>=) is improper.

`# operator should be followed by a macro argument name`

A macro argument name must be present after the # operator.

`parameter can not have any storage class other than register`

You cannot specify any storage class other than register for an argument.

`parameter cannot have initializer in function header`

You cannot initialize an argument at a function’s header segment.

`parameter in this definition incompatible with prior use`

An argument defined is inconsistent with the previous declaration.

`parameter incompatible with previous declaration`

An argument is inconsistent with the previous declaration.

`parameter missing, declaration is not allowed`

The number of arguments is insufficient (different from what is declared).

`parameter name only is not allowed on function declaration`

Giving an argument name alone in declaring a function is not permitted.

`parameter name starts with a digit in #define`

The first argument of #define starts with a digit (it must be an identifier).

`parameter redeclared in current declaration`

An argument has been redefined.

`#pragma keyword has unrecognized keyword/option`

An improper specification is present in #pragma.

Table 14.6 Error Messages (12/15)

Message	Description
`prior errors have corrupted symbol type`	The type of a symbol has been broken by a previous error.
`proscribed type specifier combination in declaration`	A declaration is present in which a given combination of types is forbidden.
`redefined statement label in function scope`	A label has been redefined in a function.
`referenced label not declared in current function scope`	A label to be referenced cannot be found in a function.
`‘register’ must appear within a function`	A declaration of a register variable is present outside a function.
`return type incompatible with function’s defined return type`	The return value of a function is different from the one declared.
`return with expression in function with “void” return type`	An attempt has been made to return a value from a function brought under a void declaration.
`same argument name used in macro` *xxx*	An identical argument is used in a macro.
`second or third operand of conditional operator has invalid type`	An improper type has been defined in a ternary operation.
`size of return type from function must be known`	The return-size of a function is unknown.
`sizeof object is of unknown length`	An object whose length is indefinite has been specified in the sizeof operator.
xxx`: : storage class incompatible with subsequent file scope declaration`	The storage class declaration is inconsistent with the next file scope declaration.
xxx`: storage class is nonvalid for formal argument`	The storage class declaration is incorrect.

Table 14.6 Error Messages (13/15)

Message	Description
struct or union must not contain member of function type	You cannot include an entity of function type either in a structure or a union.
struct used as type is incomplete or undefined	A structure is either incomplete or undefined.
struct/union improperly defined, possibly as a result of previous error(s)	Either a structure or a shared set is improper. There is a chance that an error may have occurred previously.
subscript must be positive, non-zero, integral value	An array size must be an constant of integer sequence greater than 0.
syntax error	A syntax error.
syntax error: at or near symbol *xxx*	A syntax error is present near the symbol *xxx*.
syntax error in #if constant expression:	A syntax error is present in the description of a constant expression under #if.
tag conflicts with prior struct/union appearance	A tag is inconsistent with that of struct/union previously described.
tag conflicts with prior struct/union/enum appearance	A tag is inconsistent with that of struct/union/enum previously described.
tag redeclared in current declaration	A tag re-declared is present.
xxx: tag use inconsistant with previous definition	The use of the tag *xxx* is inconsistent with the one in the previous definition.
xxx: tentative definition static object shall not have incomplete type at declaration	The type declaration of the static object is incomplete.
too many arguments to macro `***' (*n* arguments)	The number of arguments of the macro *xxx* is too many (*n* arguments are present).

Table 14.6 Error Messages (14/15)

Message	Description
too many characters in a character constant	The number of characters in a character constant is too many.
too many initializers	The number of initializers is too many.
too many parameters for ‘asm’	The number of arguments of the asm function is too many.
type cannot be inherited from a typedef	This type cannot be taken over by typedef.
type incompatible with subsequent definition	Type is not compatible with the later definition.
xxx: type incompatible with subsequent definition	The type is inconsistent with the next definition.
type of argument does not match with prototype	The type of argument does not agree with that given in the function prototype declaration.
type of ‘asm’ parameter must be integral	The argument of the asm function must be integer type.
xxx: type of initialized entity can not be function type	The type of initialized components cannot be put into the function type.
type of struct member is an array of unknown length	An array whose size is indefinite is declared on a structure’s member.
type used for this symbol is incomplete or undefined	This type of symbol is either incomplete or undefined.
unable to evaluate case label (out of range?)	The expression cannot be evaluated by case.
unbalanced #endif	Meaningless #endif has been specified.

Table 14.6 Error Messages (15/15)

Message	Description
undefined static function	A function brought under a static declaration cannot be found.
xxx: undefining not-userdefined macro	You have attempted to eliminate a macro which is not defined.
unexpected string as initializer	An unexpected character string has appeared.
union used as type is incomplete or undefined	The name of a shared set is either incomplete or undefined.
xxx: unknown preprocessor directive	A preprocessing directive is indefinite.
unnamed parameter	An argument name is missing.
unterminated character constant	The end of a character constant cannot be found.
unterminated comment	The end of a comment cannot be found.
unterminated #*xxx* conditional	The #endif directive for the #*xxx* directive is missing.
unterminated macro call	The end of a macro call cannot be found.
unterminated string	The end of a character string cannot be found.

14.2.5 Fatal Error Messages

Table 14.7 Fatal Error Messages

Message	Description
`file inclusion is too deep (up to %d nesting level)`	A file has been nested too deep.
`no memory available`	No memory available
`out of memory`	No memory available (From the optimizer.)
`prior errors have corrupted symbol scoping`	The scope has been broken by a previous error.

Appendix A

Extended Functions Reference

CC32R has additional extended functions to facilitate its incorporation in systems using the M32R family.

Appendix A describes how to use the extended functions other than those related to language specifications.

Table A.1 Extended Functions

Extended function	Description of function
Base register function	1.This function enables the code size to be minimized by specifying which of the several dedicated base registers for 16-bit register relative indirect addressing each variable is relative to (16-bit register relative indirect addressing). 2. The following are required in order to use this function: (1) Access Control File*, and (2) Compile option "-access=access_control_file".
Memory Models	1. When compiling the program, this function hypothesizes multiple storage patterns in the application's memory space so that the optimum object can be generated depending on the size and location of the code (P and C sections) and data (D and B sections) in the address space. 2. This compiler has four memory models. (1) Small model (2) Small model (with C compiler option "-memlarge") (3) Medium model (4) Large model
#pragma Extended functions	1. This extended functions can be used to efficiently access the M32R family hardware specifications from the C language.
Inline expansion	This function is such that the contents of C language function to be called are expanded directly in place of function call. Since the overhead, such as the subroutine jump instruction (BL), can be omitted, it is possible to obtain a more advantageous code in view of speed than normal function call by means of inline expansion.
M32R/ECU#5 (M32R-FPU core) Compatible Function	The new 32180 and 32182 Group (abbreviated as M32R/ECU#5) MCU's extension instructions and FPU instructions support. This function is compatible with the M32R/ECU#5
About Japanese-Kanji character processing	1. The Japanese character can be described to the character constant of a program. * The Japanese character can be processed as the multi-byte character and wide-character. 2. JIS (EUC-JP, Shift-JIS), Unicode (UTF-8) are able to be used as the character code of the Japanese character correspondence. 3. You can control those character codes flexibly by operation of the environment variable.

*. The Access Control File contains the following information:

(1)Base address for 16-bit register relative indirect addressing

(2)Register storing the base address

(3)Objects to which the base register function is applied (variables and structures)

See A.1.7, "The access Control File" for details.

A.1 Base Register Function

A.1.1 What is the Base Register Function?

The base register function enables the code size to be minimized by specifying which of the several dedicated base registers for 16-bit register relative indirect addressing each variable is relative to (16-bit register relative indirect addressing).

The code generated by CC32R consists of the following:

(1) Code using 16-bit register relative indirect addressing is generated for accessing objects allocated to the D section (area for data with initial values) and B section (area for data without initial values).

(2) Code using 16-bit register relative indirect addressing is generated for the read/write access of objects at fixed addresses.

To use this function, the following are required:

- Access Control File [*1]
- When compiling, the compile option "-access=access_control_file"

Note that the Access Control File can be generated using the "map32R" [*2] map generator.

[*1] The Access Control File contains the following information:
(1)Base address for 16-bit register relative indirect addressing
(2)Register storing the base address
(3)Objects to which the base register function is applied (variables and structures)See A.1.7, "The access Control File" for details.

[*2] For details of the 'map32R" map generator, see Part 3, "Map Generator map32R" in the "M3T-CC32R User's Manual <Assembler>".

A.1.2 Types of Access Targeted by Base Register Function, and Code Output

A.1.2.1 Access to Variables

a. Targeted variables

The variables specified on the object registration line of the Access Control File and matching A.2.2, "Objects Targeted by Base Register Function" are targeted by the base register function. (Objects defining addresses mapped by #pragma ADDRESS are treated in the same way).

b. Generated code

When outputting the code for accessing variables, the expression "variable label - base symbol" is used for the relative indirect offset.

The base symbol shows the corresponding base address, and the format is __REL_BASExx (where xx is the number of the register (11 to 13)). However, note that this symbol is not defined in the output code of the compiler (referenced by .IMPORT), and it is therefore necessary to define the value in the startup file, etc.

[Example code output]
The following is an example of the code output for base R12 and global variable var access. (In this example, var is an unsigned char type.)

Example 1) Writing

[No base register definition]	[Using base register function]
LD24 R0,#_var STB R1,@R0	**STB R1,@(_var-__REL_BASE12,R12)**

Example 2) Reading

[No base register definition]	[Using base register function]
LD24 R0,#_var LDUB R0,@R0	**LDUB R1,@(_var-__REL_BASE12,R12)**

A.1.2.2 Accessing constants

a. Targeted constants

When casting constants to the pointers shown below (A.1.3, "Objects Targeted by Base Register Function), to access an area specified by the pointer and when this address is in the range (base address -0x8000 to base address +0x7FFF) around the base address, they are the target of the base register function.

Example 1) *((int*)0x12345678) = 100; /* writing to address 0x12345678 */
Example 2) return *((int*)0x12345678); /* reading from address 0x12345678 */

b. Generated code

When outputting code for access to a constant address, the expression "constant address, base address" is used as the relative indirect offset.

[Example code output]
When the base register is R13 = 012340000 (Hex).

Example 1) Writing

[No base register definition]	[Using base register function]
SETH R0,#HIGH(0x12345678) OR3 R0,R0,#LOW(0x12345678) ST R1,@R0	**ST R1,@(0x5678,R13)**

Example 2) Reading

[No base register definition]	[Using base register function]
SETH R0,#HIGH(0x12345678) OR3 R0,R0,#LOW(0x12345678) LD R0,@R0	**LD R0,@(0x5678,R13)**

A.1.3 Objects Targeted by Base Register Function

A.1.3.1 Memory class linkage

Objects targeted by the base register function must be objects statically mapped to memory.

[Non-function objects]
(1) Global variables
(2) static global variables
(3) extern global variables
[Objects in functions]
(4) static variables in functions
(5) static variables in blocks

A.1.3.2 Object types

The following types of objects are targeted by the base register function:

(1) Integer types (char/short/long, signed/unsigned, enum)
(2) Floating-point numbers (float/double)
(3) Structures (including bit field members)

(4) Unions
(5) Arrays
(6) Pointers

A.1.3.3 Types of type qualifier

The following object type qualifiers are targeted by the base register function:

(1) No type qualifier
(2) Volatile

A.1.4 Objects Not Targeted by Base Register Function

A.1.4.1 Types of types and derived types, etc.

(1) Member names (Structure type names can be specified.)
(2) Functions
(3) Constants

A.1.4.2 Memory classes and storage

(1) static variables in functions or in blocks that have the same name as those in objects not in functions
(2) Global variables with names the same as objects in functions
(3) auto
(4) register
(5) typedef

A.1.4.3 Qualifiers

(1) const

A.1.5 Setting Base Symbols and Base Registers

With the base register function, the compiler's output code does not "(1) Definition of base symbol" or "(2) Initialization of base registers". These operations must therefore be included in, for example, the start up program.
In the example shown below, we have extracted the relevant portion of a start up program in which the three base registers R11 to R13 are used.
Note that in this example, all three registers R11 to R13 are assigned as base registers but that it is not necessary to set up all of R11 to R13 when they are not used.

[Note]
When the base address is specified in the Access Control File, you must specify the same value for the base symbol as at that address.

(1) Definition of base symbol (.export can also be .global)

```
        .EXPORT __REL_BASE11
        .EXPORT __REL_BASE12
        .EXPORT __REL_BASE13

__REL_BASE11:  .EQU  0x10000000
__REL_BASE12:  .EQU  0x20000000
__REL_BASE13:  .EQU  0x30000000
```

(2) Initialization of base registers

```
SETH    R11,#HIGH(__REL_BASE11)
OR3     R11,R11,#LOW(__REL_BASE11)

SETH    R12,#HIGH(__REL_BASE12)
OR3     R12,R12,#LOW(__REL_BASE12)

SETH    R13,#HIGH(__REL_BASE13)
OR3     R13,R13,#LOW(__REL_BASE13)
```

A.1.6 Base Register Function Limitations

(1) When the offset is 32768 or greater

With extremely large structures or arrays, the base register function cannot be used to access members or elements mapped to offsets greater than 32768 bytes from the base.

(2) Duplication of base addresses

If there are duplicate ranges covered by the base registers and objects covered by A.1.2.2, "Accessing Constant Addresses" are within the duplicated areas, the first base register with a base address defined is used to cover that area.

A.1.7 The Access Control File

The Access Control File contains the following information, which is required in order to use the base register function:

(1) Base address for 16-bit register relative indirect addressing
(2) Register storing the base address
(3) Objects to which the base register function is applied (variables and structures)

This file is specified in the compiler option "-access=access_control_file".
Note that the Access Control File can also be generated using the "map32R" map generator [*].

A.1.7.1 Contents of the Access Control File

The Access Control File contains the following:

(1) Base address

This item specifies the base address for the base register function (register relative indirect addressing).
When using fixed address access (read/write) of the area around the base address (base address -0x8000 to base address +0x7FFF), the code will be generated for register relative indirect addressing.

(2) Base register

This item specifies the register storing the base address.
Any of R11, R12, and R13 can be allocated.
Registers assigned as base registers are not used within functions for other purposes such as temporary work areas.

(3) Target objects

This item specifies the objects that are to be targeted by the base register function (16-bit register relative indirect addressing).
The addresses to which objects (variables, structures, arrays, etc.) are mapped are not decided at the time of compiling, so each object must be registered in the Access Control File (including #pragma ADDRESS).

[*]. For details of the 'map32R" map generator, see Part 3, "Map Generator map32R" in the "M3T-CC32R User's Manual <Assembler>".

A.1.7.2 The Access Control File Syntax

The Access Control File is written with each item on a new line.

(1) Comment line

[Code format]	[Format 1]	; Comment
	[Format 2]	(Blank line)
[Functions]	The comment line is ignored.	
[Example]	;This is comment 1 ← ← ;This is comment 2 ←	Comment in [format 1] Comment in [format 2] (blank line) Comment in [format 1]

(2) Base register definition line

[Code format]	@base register name [base address]
[Functions]	Defines the base address and the registers allocated as base registers
	(a) Base registers • The same register cannot be specified two or more times as a base register. (b) Base address • Omissible. If omitted, accessing fixed addresses is not available. • Specify the hexadecimal value (in 8 or fewer digits) preceded by '0x'. • Base address 0xffffffff is reserved and cannot be specified.
[Example]	@R13 0xF78000 @R12

(3) Object registration line

[Code format]	[Format 1]	Object name
	[Format 2]	Object name | source file name | function name
	[Format 3]	∗
	[Format 4]	∗ SectionName
[Functions]	One or more object registration lines can be included following the base register definition line. The object registration lines specify the names of the objects for 16-bit register relative indirect addressing using the base register specified in the preceding line. (a) Format 1: • Global variables, and static variables within the file (b) Format 2: • Variables within the specified function in the specified file (c) Format 3: • All static objects not written in other object definition lines in the Access Control File (wildcard specification). • Format 3 can only be used once in the Access Control File. (d) Format 4: ∗ Applies to all static objects belonging to the specified section that	

are not written in other object definition lines (except Format 3) of the access control file (i.e., section-specified wildcard specification).

∗ The specified section name is either (1) altered section name by #pragma SECTION or (2) designated section name. The base register is applied to the static objects that are allocated to these sections.

∗ Even if the designated section specifies a P or C section name, the specification is ignored.

∗ Specification of format 4 for the same section name can be written only once in an access control file.

[Examples]

var1 ←── [Format 1]
var2|samplw.c|func ←── [Format 2]
∗ ←── [Format 3]
∗D1 ←── [Format 4]

A.1.7.3 Hints on describing the Access Control File

- **To specify static variables in a function.**

Specifying "variable name | file name | function name" as the variable name makes static variables within that function the subject of the base register.

Example: var2|sample.c|func

This specifies variable var2 in function func in source file sample.c. This does not apply to var2 not belonging to the specified function.

- **To specify all variables together.**

You can specify the wildcard "*" for the variable name to include all variables not otherwise specified that are allocated to the data area (sections D and B).
Example: The following example shows the lines from the Access Control File that assigns base register R13 to global variable var1 and base register R12 to all other variables in the data area (D and B sections).

```
Example:
@R13      0x00F78000
var1
@R12      0x00F88000
*
```

- **Comments**

(1) Lines starting with the semicolon (;) are processed as comments.
(2) If a blank is encountered after the name of a variable, the rest of that line is processed as a comment. (Note, however, that the bar (|) cannot be included in comments.)

- **Blanks**

  (1) Tabs are processed as blanks.

  (2) Successive blanks at the beginning of a line are ignored. (This also applies to blanks before a semicolon.)

- **Duplicate specifications**

  The following cannot be duplicated:

  - ◆ Identical base register names
  - ◆ Identical variable names
  - ◆ Wildcard specifications

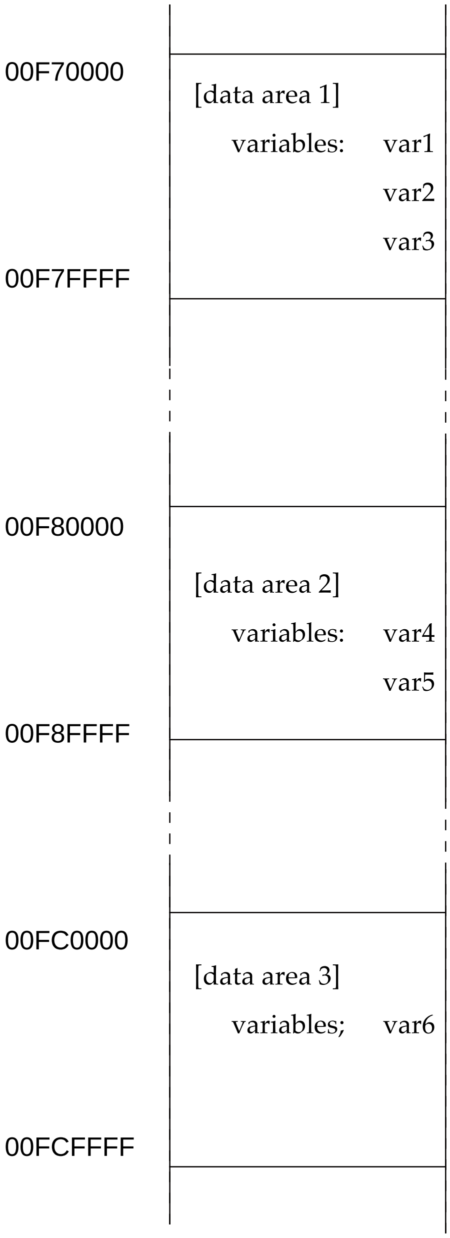
A.1.8 Example of Using Base Register Function

This section describes how to use the base register function.

A.1.8.1 Example Use of Base Register Function

As shown in the figure below, there are three 64KB data areas. Explain the basic procedure ([1] to [5] below) for allocating a base register to each of these data areas.

- In the example, base registers R13, R12, and R11 are allocated respectively to data areas 1, 2, and 3.
- Additionally, data area 1 accommodates global variables var1, var2 and var3, data area 2 accommodates global variables var4 and var5, and data area 3 accommodates global variable var6.



[1] Determining the base address

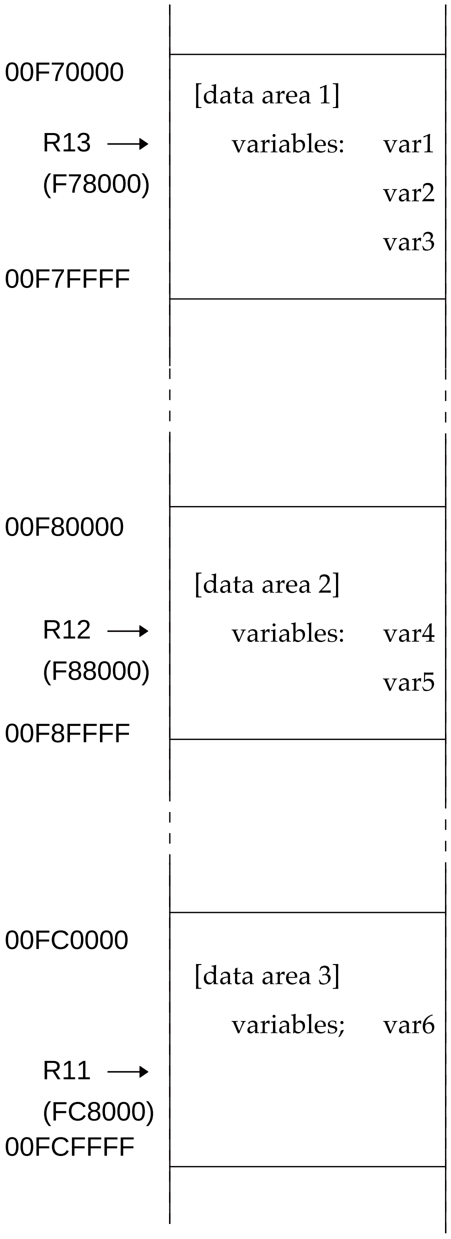
The base address is a fixed address set in the base register.
Because of the M32R register relative indirect specifications, the base registers can cover the following range:

Base address -0x8000 to base address +0x7FFF.

The base addresses for covering data areas 1 to 3 are as follows:

data area 1	0xF78000
data area 2	0xF88000
data area 3	0xFC8000

These are allocatde to R13, R12, and R11, respectively.



[2] Creating the Access Control File

After determining the base addresses, create the Access Control File, which defines the details of the base register function. (Assume the file name is sample.acc.)

sample.acc

```
@R13  0xF78000
var1
var2
var3

@R12  0xF88000
var4
var5

@R11  0xFC8000
var6
```

As shown above, the base register is defined using the following format:

'@' base_register_name   '0x' base_address (hex)

The global variables to be covered by that base register are listed on the following lines.

- The Access Control File can also be automatically generated from the load modules. See Section 2, "map32 Map Generator" in the "CC32R User's Manual <Assembler>" for details.
- For details of how to specify static variables in a function, see "A.1.7.3, Hints on describing the Access Control File."

[3] Compiling

Compile the program specifying the Access Control File created in step [2].
Add -access=sample.acc to the compiler command line.

[Example output code]

The following shows the code for var1 and var4 resulting from compiling the program with sample.acc specified and var1 and var4 as int type global variables.

- C language

```
(Expression 1)    var1 = 5;       /* writing var1 */
(Expression 2)    return var4:    /* reading var4 */
```

- Output code when applying base register function

```
(Expression 1)    LDI   R0,#5
                  ST    R0,@(_var1-__REL_BASE13,R13)

(Expression 2)    LD    R0,@(_var4-__REL_BASE12,R12)
```

[4] Defining the base symbols and setting the base registers

Next, add the code for (a) defining the base symbols, and (b) setting the base registers in the startup program.
The base address is represented by the base symbol __REL_BASExx (where xx is the base register number. This symbol is required for calculating the offsets when generating the code. (See "Example output code" in step [3], "Compiling.")
In the following example, the settings are for base registers R13, R12, and R11.

(a) Definition of base symbols
(b) Setting base registers

[Note]

- As with the Access Control File, a base register setting program can also be automatically generated by map32R. See Section 2, "map32 Map Generator" in the "CC32R User's Manual <Assembler>" for details.

[5] Linking

Link the component programs, including the program created in step [4].

A.2 Memory Models

A.2.1 About Memory Models

This compiler has four memory models available, helping to develop your application programs efficiently.
Memory models refer to several assumed patters for applications to be stored in address space, provided to ensure that optimum objects will be generated according to the size and position of address space in which code (sections P, C) and data (sections D, B) are stored when compiling.
This helps to generate the most suitable object for the size of each application developed.

The four memory models available for this compiler are shown below.

Memory model name	Address space usable for code and data	C standard library
Small model	Code 0x00000000 to 0x00FFFFFF Data 0x00000000 to 0x00FFFFFF	m32RcRlib
Small model (with -memlarge attached)	Code 0x00000000 to 0x00FFFFFF Data 0x00000000 to 0xFFFFFFFF (Entire 32-bit memory space)	m32RcRM.lib
Medium model	Code Given address A to (A + 0x00FFFFFF) Data 0x00000000 to 0xFFFFFFFF (Entire 32-bit memory space)	m32RcRM.lib
Large model	Code 0x00000000 to 0xFFFFFFFF (Entire 32-bit memory space) Data 0x00000000 to 0xFFFFFFFF (Entire 32-bit memory space)	m32RcRL.lib

❋ When using C standard libraries, be sure to use the library file that corresponds to each memory model.

A.2.2 Detailes of Memory Models

Each memory model is detailed below.

- Small model.

The small model is a memory model in which both code and data of the application are assumed to be stored within the address space of
0x00000000 to 0x00FFFFFF (colored shaded part in Figure 3.3). To compile the source in this memory model, specify the option shown below when compiling

```
-small
```

If no memory model is specified, the compiler by default assumes the small model as it compiles the source.
The C standard libraries that correspond to this memory model are:

m32RcR.lib (for functions passed via register)

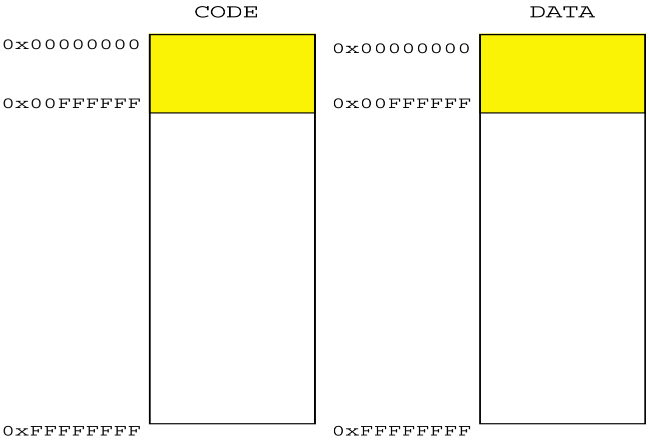


Figure A.1 Address Space in Small Model

- Small model (with -memlarge attached)

The small model (with -memlarge attached) is a memory model in which the code and the data of the application respectively are assumed to be stored within the address space of

0x00000000 to 0x00FFFFFF (colored part in Figure A.2)

and the address space of

0x00000000 to 0xFFFFFFFF (colored part in Figure A.2).

To compile the source in this memory model, specify the option shown below when compiling

-small -memlarge

The C standard libraries that correspond to this memory model are:

m32RcRM.lib (for functions passed via register)

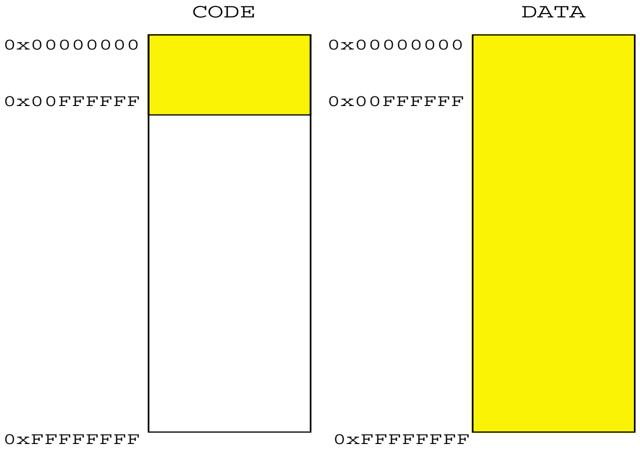


Figure A.2 Address Space in Smal(with -memlarge attached)l Model

- Medium model

The medium model is a memory model in which the code and the data of the application respectively are assumed to be stored within the address space of

given address A to A + 0x00FFFFFF (colored part in Figure A.3)

and the address space of

0x00000000 to 0xFFFFFFFF (colored part in Figure A.3).

To compile the source in this memory model, specify the option shown below when compiling

-medium

The C standard libraries that correspond to this memory model are:

m32RcRM.lib (for functions passed via register)

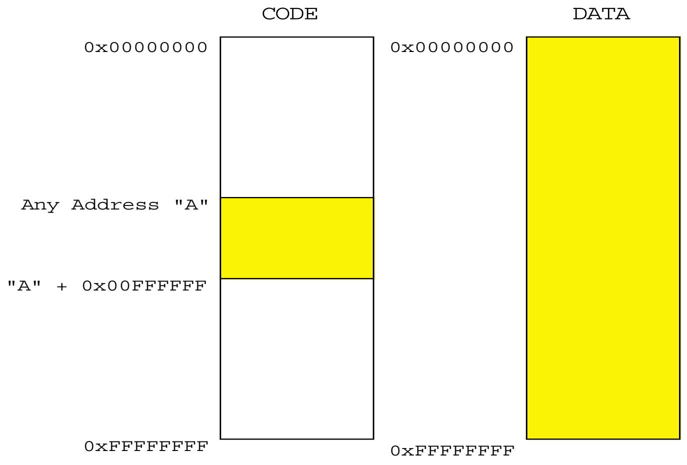


Figure A.3 Address Space in Medium Model

- Large model

The large model is a memory model in which both code and data of the application are assumed to be stored within the address space of

0x00000000 to 0x00FFFFFF (colored part in Figure A.4).

To compile the source in this memory model, specify the option shown below when compiling

-large

The C standard libraries that correspond to this memory model are:

m32RcRL.lib (for functions passed via register)

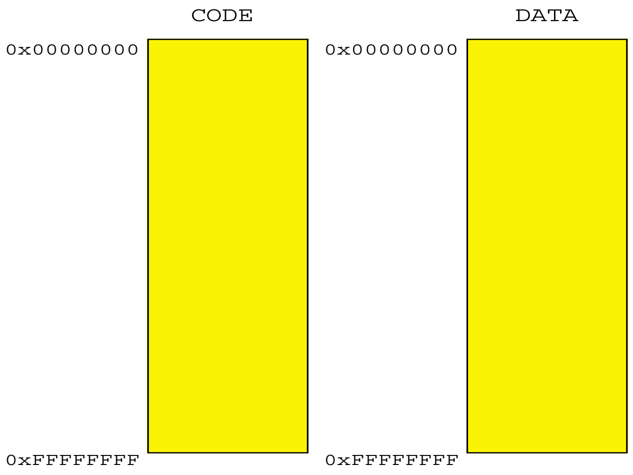


Figure A.4   Address Space in Medium Model

A.3 #pragma Extended Functions

A.3.1 List of #pragma Extended Functions

The following table lists the contents and rules pertaining to the extended functions related to #pragma.

Table A.2 List of #pragma Extended Functions

Extended function	Description of function
#pragma ADDRESS	Declares that the specified variable is mapped to the specified absolute address. Syntax : #pragma ADDRESS *variable-name∆absolute-address* [1] Example : #pragma ADDRESS val_1 0x1000
#pragma SECTION	Changes the default section name created by the compiler. Syntax : #pragma SECTION *default-section-name∆modified-section-name* [1] Example : #pragma SECTION B USR_SEC_B
#pragma INTERRUPT (#pragma INTF)	Declares an interrupt function described in C. This declaration generates the code for processing of an interrupt processing function at the entry and exit points of a function. Additionally, a return is performed by the RTE instruction at the exit of the function. Syntax : #pragma INTERRUPT interrupt-function∆[register -name∆register-name ...][1] Example : #pragma INTERRUPT int_func R6
#pragma ketword asm on #pragma keyword asm off	Declares the use of the in-line assembly function (asm function) [2] This setting is used so that asm is interpreted not as a normal identifier but as a keyword. Specify #pragma keyword asm off to restore asm to a normal identifier. Syntax : #pragma keyword asm *on* (or *off*) Example : #pragma keyword asm on

[1] “∆” is space code (mandatory).

[2] For details of the 'The in-line assembly function", see Section 7.4.

#pragma ADDRESS

Function to specify absolute address of variable

Function : Declares that a variable is mapped to the specified absolute address.

Syntax : #pragma ADDRESS *variable-name∆absolute-address*

Description :

●#pragma ADDRESS declares that a variable is mapped to a specified absolute address.

●The compiler generates the code to define the variable name specified in this declaration as the value of the symbol in the assembler pseudocommand .EQU. Therefore, the area for the variable is not secured.

●The absolute address is output as a character string in a .EQU pseudocommand expression. Therefore, the coding rules for the assembler .EQU pseudocommand apply. Note that only the following numerical format constants can be specified for the absolute address. Also, the expression can only be described as a constant expression.

Octal	: A constant starting with 0 and consisting of 0 to 7 numerals.
Decimal	: A constant starting with other than 0 and consisting of 0 to 9 numerals.
Hexadecima	: A constant starting with 0x or 0X and consisting of numerals 0 to 9 and A(a) to F(f).

●Variable names of the following storage classes and data types can be specified for the variable name.

[Storage classes]

Externally definition variables (global variable)
Externally referenced variables (extern variable)
Static definition variables (static variables excluding static variables defined within a function)

[Data types]

char, short, int,long
unsigned char, unsigned short, unsigned int, unsigned long,
long double, float, double,
struct, union
enum,pointers

●The #pragma ADDRESS declaration applies to variables defined both prior to and subsequent to this declaration.

Rules :

●If #pragma ADDRESS is declared two or more times for the same variable name, error is returned at compiling.

●Variables for which an initialization expression has been described cannot be specified. If they are specified, a warning is returned at compiling and the declaration has no effect.

●Variables (local variables, and static variables defined within a function) defined within a function cannot be specified. If they are specified, a warning is returned at compiling and the declaration has no effect.

●Structure member names, union member names, enum-type member names, or array elements cannot be specified as variable names. If they are specified, the declaration has no effect (no warning is displayed).

●Function arguments cannot be specified as variable names. If they are specified, the declaration has no effect (no warning is displayed).

#pragma ADDRESS

Notes:

- No check is performed to see if the specified absolute address follows the variable alignment. Be sure to specify the absolute address following the alignment of data type.
- No check is performed to see if the variable areas overlap as a result of the specified absolute address. Take care when specifying the address.
- No check is performed to see if array or structure access addresses overflow the 32-bit address as a result of the specified absolute address. Take care when specifying the absolute address that there is no overflow.
- The specifying of the same absolute address in a #pragma ADDRESS declaration for a different variable has an effect on the compiler's optimization processing and correct code may not be generated. When specifying the same absolute address, use the volatile qualifier as a variable.
- Note that the following will result in faulty operation but are not checked by the compiler.

[Example of specifying addresses that overlap for variable porta and portb areas]

```
#pragma ADDRESS porta 0x10010
#pragma ADDRESS portb 0x10012
int porta;
int portb;
```

[Example of specifying overflow array access address]

```
#pragma ADDRESS x 0xffffffff
char x[2];
void func(void)
{
      x[1] = 0;
}
```

[Example of effect on compiler's optimization processing]

```
#pragma ADDRESS x 0x1000
#pragma ADDRESS y 0x1000
int x, y;
int func(void)
{
      x = 0;
      y += 1;*
      if (x == 0)
            return 0;
      return 1;
}
```

* Under the ANSI-C specifications, the interpretation of modification (y += 1) of y, above, does not affect the value of x, and is allowed. Thus, the if statement x == 0 can be interpreted as always true, and the compiler might generate code that always returns 0.

Figure A.5 Examples of #pragma ADDRESS Coding Not Checked by Compiler

#pragma ADDRESS

Examples: ●#pragma ADDRESS is written directly into the C source code, as follows:

[Example coding:]

```
#pragma ADDRESS var1    0x10000
#pragma ADDRESS var2    0x20000
char var1;
static char var2;

void
func(void)
{
      var1 = var2;
}
```

[Output from example coding (assembler code]

```
_var1:      .EQU  0x10000
            .SECTION    P,CODE,ALIGN=4
            .EXPORT     $func
$func:
            LD24  R1,#0x20000
            LD24  R0,#0x10000
            LDB   R1,@R1
            STB   R1,@R0
            JMP   R14

            .EXPORT     _var1
            .END
```

Figure A.6 Example Use of #pragma ADDRESS declaration

In this example code, var1 is mapped to address 0x10000 and var2 is mapped to address 0x20000.

By so doing, code is generated such that 1 byte is read from address 0x20000 and written to address 0x10000 by the expression var1 = var2 in func().

Also, because var1 is an externally defined variable, .EQU defines the corresponding symbol "var1" and .EXPORT makes it possible for other objects to reference it

#pragma SECTION

Function to change section name

Function : Declares that a variable is mapped to the specified absolute address.

Syntax : #pragma SECTION *default-section-nameΔmodified-section-name*

Description :

●#pragma SECTION changes the default section name created by the compiler to a user-defined section name. This declaration is valid until another #pragma SECTION declaration for the same section is encountered or the end of the file.

●Specify one of the following section names created by the compiler in default section name.

Section Name	Section Attribute	Location Attribute	Description
P	CODE	ALIGN=4	Program code area
C	DATA	ALIGN=4	Constant data area
D	DATA	ALIGN=4	Data area with initializers
B	DATA	ALIGN=4	Data area without initializers

●Specify a new user-defined section name in modified section name. The rules of description for section names depend on the rules of description for the assembler names (see 3.5, "The rules of description for names" in the "User's Manual <Assembler>"). The section attributes (section attribute and location attribute) of the modified section name are the same as the attributes of default section name. The following is extracted from section 3.5, "The rules of description for names" in the "User's Manual<Assembler>".

> Rules for names are given below :
>
> ■Characters you can use for the leading character
> One of alphabetic letters, dollar sign ($), and underscore (_).
> You cannot use a digit for the leading character.
>
> ■Characters you can use for the second and subsequent characters
> One of alphanumeric characters, dollar sign ($), and underscore (_)
>
> ■The number of characters you can use in a name
>
> | | |
> |---|---|
> | Module name | : 206 characters |
> | Symbol name | : 243 characters |
> | Section name | : 243 characters |
> | Preprocessing name | : 32 characters |
> | Macro name | : 32 characters |
>
> ■Distinction between uppercase and lowercase letters
> Distinction is made for names the user can define.
> No distinction is made for names the user cannot define.
>
> You define a name according to the preceding rules. Be careful about the following in that instance :
>
> ■You cannot use a name identical to a reserved word for an entity other than a module name.
>
> ■Names the user defines, such as symbol names, section names, cannot be duplicated.

#pragma SECTION

Description :

- The default compiler section name can be restored by specifying the default section name in modified section name.
- To change the section name, declare the #pragma SECTION before the function definitions or data definitions.
- The D section must be initialized to the initial values at startup. If you change the name of the D section, be sure to initialize the modified section to the initial values at startup.
- The B section must normally be cleared (to zeros) at startup. If you have changed the name of the B section, be sure to clear the modified section (to zeros) at startup.

Rules:

- If you change the name of the P section, section names in functions defined after this declaration take the modified section name.
- If you change the names of the C, D, or B sections, section names for variables and constant data defined after this declaration take the modified section name. However, this is only valid until a delimiter (comma ",", semicolon ";", or equal sign "=") after a variable name for which data has been defined.
- This declaration can be made multiple times for the same default section.
- The section attribute or location attribute cannot be specified.
- If #pragma SECTION is used together with the "-R" compiler command line option ("-R = default section name=modified section name", which changes the section name created by the C compiler):
  - The #pragma SECTION declaration is not affected by the -R command line option. If both are used, the #pragma declaration takes precedence where the #pragma SECTION declaration is valid.
  - If modified section name in the #pragma SECTION declaration is the same as the default section name (if the default section name generated by the compiler is restored), the command line option '-R" is valid.

Notes:

- You cannot split one function or one data definition into two or more section names.
- If a C source program with a renamed P section is compiled using the debugging information ("-g" option), a P section with a zero size is created as a result of the object format specifications. This has no effect on program execution.
- Use the same section name #pragma declaration for the same variables. Do not specify a different section name in a temporary definition, etc.

#pragma SECTION

Examples: ●#pragma SECTION is directly written into the C source code, as follows:

[Example coding:]

```
int a;
#pragma SECTION B B1
int b;
#pragma SECTION B B2
int c;
#pragma SECTION P P1

void func(void)
{
   ...
}
```

Variable a is mapped to section B (because it occurs before the #pragma SECTION declaration).

Variable b is mapped to section B1.

Variable c is mapped to section B2.

Function func is mapped to section P1.

Figure A.7 Example Use of #pragma SECTION Declaration

#pragma INTERRUPT (#pragma INTF)

Function to describe interrupt-processing function

Function : Declares an interrupt processing function written in C.

Syntax : #pragma INTERUPT *interrupt-processing-function-name*Δ[*register-name*Δ*register-name*Δ...]*

#pragma INTF *interrupt-processing-function-name*Δ[*register-name*Δ*register-name*Δ...]*

Description :

●#pragma INTERRUPT declares a specified function as an interrupt processing function.

● The compiler generates the code for evacuating and returning the register to be used to the stack at the entry and exit points of the function declared as the interrupt processing function. It uses the RTE instruction to execute a return at the exit of the function.

●If no register name is specified (default setting), the compiler generates the code for evacuating and returning registers R0 to R7 and register R14 (if there is a function call within the interrupt processing function).

Because, due to the function call rules (see 6.3, "Basic Procedures for Function Call and Return " in the "User's Manual"), registers R8 to R13 are reserved, they are not evacuated or returned. (Even if any of R11 to R13 are specified as base registers, they are similarly, by default, not evacuated or returned.) The following is extracted from 6.3, " Basic Procedures for Function Call and Return " in the "User's Manual."

> (4)Save registers
> The contents of registers (R4-R13) to be used by the called function are saved on the stack.
> The size of the saving area is ìthe number of registers to be saved ×
> 4 bytesî. When the link register R14 is used, then the link register is saved.

● The following registers can be specified for register names:

General registers: R0,R1,R2,R3,R4,R5.R6,R7,R8,R9,R10,R11,R12,R13,R14
Control registers: PSW (or CR0), BPC (or CR6)
Accumulator: ACC0 (or A0), ACC1 (or A1), ACC

When a register name is specified, it is added to the default setting and the compiler generates the code for evacuating and returning that register. To have other than the default registers evacuated and returned, register names must be specified. To specify two or more registers, specify their names delimited by spaces. Note that there is no differentiation between uppercase and lowercase letters when specifying the names.

Notes: 1. ACC1 (or A1) is a reserved register name for the M32Rx core instruction set, and this specification is ignored.
2. If ACC is specified, the operation is the same as when only ACC0 (or A0) is specified.

●The #pragma INTERRUPT declaration is valid whether the actual function is defined before or after the declaration.

●To ensure compatibility with other compilers, it is able to describe #pragma INTF instead of #pragma INTERRUPT.

●Base registers (R11 to R13) specified using the -access option are not evacuated or returned unless specified in a #pragma INTERRUPT statement.

#pragma INTERRUPT

Rules:

- An error is returned when compiling in the case of interrupt processing functions other than those of the void type and an interrupt processing function that has arguments are described.
- An error is returned when compiling if a #pragma INTERRUPT is declared two or more times for the same function.
- If other than a function name is specified, that specification is ignored (#pragma INTERRUPT is invalid).
- The compiler does not generate instruction code for reenabling the interrupt within the interrupt processing function.
- The compiler does not handle multiple interrupts (no code is generated for reenabling the interrupt within the specified function).

Examples: #pragma INTERRUPT is written directly into the C source code as follows:

[Example of use:]

```
#pragma INTERRUPT int_func R12
extern int counter;
void int_func(void)
{
        counter ++;
}
```

In this example, the #pragma INTERRUPT specifies function int_func as the interrupt processing function and the compiler generates the code for evacuating and returning the default registers (R0 to R7) as well as the specified register R12. Also, instruction RTE is executed at the end of the function.

#pragma keyword asm on (#pragma keyword asm off)

Function to declaration of using in-line assembler

Function : Declares the use of the in-line assembly function (asm function)

Syntax : #pragma keyword asm on
#pragma keyword asm off

Description :

●#pragma keyword asm on (off) causes "asm" to be interpreted as an asm keyword rather than as the normal identifier. This enables the in-line assembly function to be used. To have asm again used as an identifier, specify #pragma keyword asm off. See 7.4, "In-line Assembly function" for details of the in-line assembly function.

A.4 Inline expansion

This function is such that the contents of C language function to be called are expanded directly in place of function call. Since the overhead, such as the subroutine jump instruction (BL), can be omitted, it is possible to obtain a more advantageous code in view of speed than normal function call by means of inline expansion.

[CAUTION]

When the size and scale of a function to be expanded are large, and there are many areas to be called, this expansion may adversely be affected; for example, the code size increases. It is, therefore, necessary to use this inline expansion function after satisfactorily checking this effect.

[Format]

inline Δ type-specifier Δ function(...);   (Δ means one or more white space characters.)

Example 1: Function declaration)

```
inline int func1(void);
```

Example 2: Function definition)

```
inline static int func2 (int a)
{
    a *= 2;
    return a;
}
```

[Compile Option]

1. " -O <level> (<level> = 0 to 7) "

Where the optimization exceeding Level 4 is effective, the inline expansion is made effective.

In the event of Levels 0 to 3 without optimization, no inline expansion is valid.

[NOTE]

When "-Ospace and -Otime" are designated, "-O7" becomes effective in the default even if "-O<level>" is not specified; therefore, the inline expansion becomes valid.

2. " -noinline "

The inline keyword is made invalid. The inline keyword, if specified, is ignored even when it is described. (This is not a error.)

[Explanation]

1. The inline keyword designates the specified function as the inline function (a function of applicatable to inline expansion).
   Also, designate the inline keyword when the function is defined. Even when it is specified for function declaration, it does not recognize this function as inline function in the absence of inline keyword when the function is defined.

2. The inline keyword, even if specified, is ignored where "-noinline" option is designated at the time of compiling. (This is not a error.)

3. Since inline expansion is carried out as one optimizing, it should be carried out only when -O4 level optimizing is effective.
4. Even in the event of forward reference of inline function, inline development is carried out.
5. Even when the nested inline function is called (the inline function calls another inline function), inline expansion is carried out, except for the inline function corresponding to the Precautions "2. Where inline development is suppressed" described later.

[Sample Use]

[Source file]

```
% cat sample.c

inline int func (int a)
{
   int b = a * 2;
   return b;
}
int answer;
void foo(int a)
{
   answer = func(a);
}
```

[Command line]

```
% cc32R -S -O7 sample.c
```

[Code to be generated]

```
% cat sample.ms

   .IMPORT $_100_builtin_memcopy
   .EXPORT _answer
   .SECTION P,CODE,ALIGN=4
   .EXPORT $func
$func:
   MV      R0,R4
   SLLI    R0,#1
   JMP R14
   .EXPORT $foo
$foo:
   MV      R0,R4   ; --- Inline expansion
   SLLI    R0,#1   ;   (Equal to "answer = a*2;")
   LD24    R1,#_answer
   ST      R0,@R1
   JMP R14
   .SECTION B,DATA,ALIGN=4
   .ALIGN 4
_answer:
   .RES.B 4
   .END
```

[Precautions]

1. Static declared inline function
   Where static declared inline function calling was all inline expanded or it was not originally called from any place, the compiler deletes its function body by judging it as unnecessary.

2. Where inline expansion is suppressed
   For the following inline function call, no inline expansion is carried out:

   [a] Variable arguments are included.

   [b] The static variable is included in the function.

   [c] The type of actual argument in function calling is not compatible with that of dummy argument.

   [d] The calling and function definition are not described on the same compilation unit.

   [e] The function calls itself. (Recursive call)

3. Debugging information
   The debugging information (C-source and generated code related information) in the inline expanded part is generated from the contents of inline function body before inline expansion. For example, if the inline expanded part is step executed with the debugger, such as PD32R, the source line of inline function body before inline expansion is displayed. However, in the following cases, the corresponding debugging function is to be restricted.

   [Restrictions to debugger during inline expansion]
   (The command name is associated with PD32R and PD32RSIM.)

   [a] Step execution.
   The line near the function entrance or exit and the line near the place where its function was called may be inversed in sequence.

   [b] Function call status display ("where" command).
   Among the functions in the calling path, the inline expanded function is not displayed.

   [c] Specified function (func command).
   For the inline function with static declaration, the original function that has not been required any more is deleted. In this case, no command for referencing the function can be used.

   [d] For the inline expanded function, avoid the following operation; otherwise, an unexpected result may be reached.

   [Assignment and reference of variable]
   Where the same name exists in the variable (including the dummy argument) inside the inline expanded function and the expansion destination (calling source), the debugger cannot discriminate between the both. In such a case, the value of these variables should not be changed and referenced.

   [Break point in source line]
   Where the source line is inline expanded and several codes are made,

only one of these codes is effective in source line break. No other source lines break are disabled.

4. Inline Development Compatibility
   Except the item not defined yet with ANSI-C, the program execution results are coincident between inline expansion and non-inline expansion.
   However, it should be noted that for the item not defined yet with ANSI as shown in the following example, the results may change, depending on inline expansion.

[Example]

For the local character string literal within the function, a new area is maintained at each inline expansion. Consequently, the results may differ, depending on inline expansion and non-inline expansion of function for rewriting the local character string literal.
(Rewriting the character string literal is not recommendable, which is not defined yet in ANSI-C.)

[Source file]

```
char c;
inline char * func(void)
{
   char *s = "abcde";
   c = s[2]; /* Set s[2] before rewriting to "c" */
   s[2] = '-'; /* Rewrite the character string */
            /*literal here */
   return s; /* Return the present pointer */
}

char c1, c2, c3, c4;
void foo(void)
{
   char *s;
   s = func();
   c1 = c;
   c2 = s[2];
   s = func();
   c3 = c;
   c4 = s[2];
}
```

[Function foo() execution results]

```
In the case of non-inline expansion of func()
(Operation for the same "abcde")
      C1: 'c'
      C2: '-'
      c3: 'c'
      c4: '-'
In the case of inline expansion of func()
(Operation for different "abcde")
      C1: 'c'
      C2: '-'
      c3: '-'
      c4: '-'
```

A.5 M32R/ECU#5 (M32R-FPU core) Compatible Function

The new 32180 and 32182 Group (abbreviated as M32R/ECU#5) MCU's extension instructions and FPU instructions support.
This function is compatible with the M32R/ECU#5.

[a] A code can be generated, using M32R/ECU#5 added instruction. (C Compiler)

[b] The program using M32R/ECU#5 added instruction can be assembled. (Assembler)

[c] The floating point constant can be handled. (Assembler)

A.5.1 Option designation

-m32re5	A code is generated, using the M32R/ECU#5 extension instruction. Also, the non-normalized numeral in the floating point constant is reduced to "0.0".
-fminst	A code is generated, using FMADD (Floating-point multiply and add operation instruction) and FMSUB (Floating-point multiply and substract operation instruction). This option is disregarded where no "-m32re5" option is valid. Where this option is not specified, the FMADD and FMSUB instructions are not used. Where this option is used, also refer to "A.5.3.4".
-float_only	The double type is regarded forcedly as the float type. If this option is used together with the "-m32re5" option, all floating point operations can be made applicable to the FPU instruction. In utilizing this function, be sure to refer to the contents given in "A.5.3.1 <CAUTION> -float-only Option" as well.

A.5.2 Utilize FPU Instruction Effectively

The "-m32re5" option generates the FPU instruction for float type operation. To exhibit M32R/ECU#5 capacity to the maximum in floating-point operation, make arrangements so that as many floating-point operations as possible are of a float type by using the following method.

Even if the "-m32re5" option is effective, the run-time routine is used as usual for the double type operation.

[1] Change the double type declaration to the float type one:
However, even in the float type, ensure that there is no problem in computing accuracy and effective range of floating-point value.

[2] Specify the precision in the floating-point constant:
Carry out precision designation (f) indicating the float type with the real arithmetic

constant suffixed by "f" as shown below.

1.234f	The suffix "f" denotes the float type
1.234	Since no precision is specified, this real constant indicates the double type.

[3] Declare the argument in the float type in prototype declaration:

Where the floating-point numerical value is used for the function return value and argument, be sure to carry out prototype declaration with type declaration in the argument.

```
Example)
  extern void func1(float);
  void func2(floa fv)
  {
      func1(fv);
  }
```

* In the absence of prototype declaration of func1 function, the argument is of a double type.

[4] Do not use variable arguments:

Since the float type covering the variable arguments is transformed to the double type, see to it that no float type is handled in variable arguments.

A.5.3 Precautions in Utilizing FPU Instruction

A.5.3.1 [CAUTION] "-float_only" Option

The "-float_only" option is such that the FPU instruction is applicable to all floating-point operations even when the source files as described. If used, however, give the greatest possible care thereto since there arises a problem of compatibility as described below.

[a] It is not match to ANSI-C standard.

[b] Function name replacement

In math.h, if the -float_only option is specified when compiling the source file, calls to the double-precision mathematic functions are changed to calls to the equivalent functions in the single-precision mathematic function.

```
Example 1: When "-float_only" is specified at the same time
       #include <math.h>
       ans = cos(rd);

           ↓ Compiled with -float_only added

       ans = cosf(rd);  /* Becomes equivalent to this */
```

This replacement for the cos function case, for example, is accomplished by a macro like the one shown below.

```
#define cos cosf
      :
(Defined the same way for other double-precision
 mathematic functions)
      :
```

The function name that is stored in the load module when a called function name is replaced, is the replaced function name and not the pre-replacement function name. Therefore, pre-replacement function names can be neither specified nor displayed in the debugger (e.g., M3T-PD32R) or TM inspector. (You can specify or display replaced function names, though.)

[c] The interface specifications for the function having the double type argument and variable arguments (that may possibly handle the floating-point value) change. Consequently, where the object generated by compiling it with the "-float_only" option and the object corresponding to any one of the following are interfaced with the function having the double type argument or variable arguments, the both cannot be linked.

- Object generated without "-float_only".
- Object generated with V.3.20 or earlier CC32R.
- Standard library (math.h, printf & scan functions, etc.)

(All floating-point operation functions of the standard library are of a double type.)

A.5.3.2 Not normalized

[a] At compiling:
Where the -m32re5 option is effective, a warning is issued when the floating-point constant is not normalized, then it is reduced to "+0.0".

[b] During execution:
Where the floating-point constant is not normalized during FPU instruction operation, there may arise non-packaging exceptions. (For FPU instruction non-normalized constant handling, refer to the M32R/ECU#5 software manual.)

A.5.3.3 Round-off mode

The CC32R is designed on the premise that the round-off mode is rounding to the nearest in float operation; it is, therefore, necessary to allow the M32R/ECU#5 rounding mode to match this mode as well.

A.5.3.4 "-fminst" Option

The FMADD instruction differs in round-off handling from the FMUL and FADD combined for computing. (This is also the same with the FMSUB and FMUL & FSUB combinations.)

It should, therefore, be noted that there may arise a difference in results between the cases where the -fminst option was designated and not designated.
(For the FPU instruction rounding mode, refer to the M32R/ECU#5 Software Manual.

A.6 About Japanese-Kanji character processing

* The Japanese character can be described to the character constant of a program.

* The Japanese character can be processed as the multi-byte character and wide-character.

* JIS (EUC-JP, Shift-JIS), Unicode (UTF-8) are able to be used as the character code of the Japanese character correspondence.

* You can control those character codes flexibly by operation of the environment variable. For example case that the C source file is written by Shift-JIS and the corresponding object shall be outputted by UTF-8.

A.6.1 character sets and character code

A.6.1.1 The Japanese character

The character set that is defined with JIS(The Japanese Industrial Standard) X 0201 and also JIS X 0208 can be used.
(Yet, the Latin character part of JIS X 0201 considers as ASCII and process.)

JIS X 0201 ... Generally it is called a "Hankaku Characters".

JIS X 0208 ... The Japaneze-Kanji character of the 1st JIS standard, the 2nd standard are included. Both those characters are called "Zenkaku Characters" generally.

In this chapter, if a character is called 'Japanese character', it includes right half of JIS X 0201 characters and all of JIS X 0208 characters.

A.6.1.1 kind of character code

The character code name that corresponds is shown in Table A.3.
In this chapter, These character codes are expressed with the name that is written to the "Character code name" column in Table A.3.

Table A.3 The chart of corresponding Japanese-Kanji character codes

Character code name	General notation	Outline	Able to be used ?		
			C source	Multi-byte character string	wide character string
euc	euc-JP	JIS X 0201,0208 EUC(Extended Unix Code) encoding	Yes	Yes	Yes
sjis	Shift-JIS	Shift-JIS encoding of JIS X 0201,0208	Yes	Yes	Yes
utf8	UTF-8	The UTF-8 encoded Unicode (UCS-2) that was transformed from JIS X 0201,0208	Yes	Yes	No
	UTF-16	The UTF-16 encoded Unicode (UCS-2) that transformed from JIS X 0201,0208	No	No	Yes

Pay attention to several points on the using of the Unicode (utf8) application.

As for the details, refer to A.6.4.7.

A.6.1.3 Method of selecting character codes

Designate the name of the character code column of Table 9 to each for the environment variable (for compiler) and the setlocale function (LC_CTYPE category, for standard library) to select the character code code that wants to use it.
Furthermore, please pay attention to the difference - the environment variable does not distinguish between the upper and lower of character code name, but setlocale function distinguishes them.

a. Character code at the time of input (C Compiler)

Environment variable [M32RKIN]

The character code that describes the C source file is designated.

* The name that shows it to "character code name" of Table A.3 is designated.
* The upper and lower have the same meaning in this environment variable.
* If this environment variable is undefined, sjis (in PC version) or euc (in EWS version) is selected in the default.

b. Character code at the time of output (C Compiler)

Environment variable [M32RKOUT]

This environment variable teaches the character code which is best for outputting character constant and string letteral to the compiler.

* The name that shows it to "character code name" of Table A.3 is designated.
* The upper and lower have the same meaning in this environment variable.
* If this environment variable is undefined, sjis (in PC version) or euc (in EWS version) is selected in the default.

c. Character code (the standard library) at the time of implementation

locale [LC_CTYPE category]

* The name that shows it to "the character code name" of Table A.3 is designated.
* The upper character or lower character are distinguished in this locale name.
* The character code is set to LC_CTYPE even in the case that the character code name is designated to the LC_ALL category.
* The initial value of the LC_ALL category is "C".
* This designation is effective to the multi-byte processing functions, printf series functions and scanf series functions.
Please pay attention because it is ineffective to string.h and ctype.h function group. As for the details, refer to A.6.4.3.

A.6.2 Description method of the Japanese character

The Japanese character can be described to the character constant and character string letteral directly.
Also, there are 2 kinds of a wide character (or a string) and multi-byte character about whether 'L' is attached or not.

- ● "漢" ... Multi-byte character (char *)
- ● L'漢' ... Wide character (wchar_t)
- ● "漢字" ... Multi-byte character string (char *)
- ● L"漢字" ... Wide character string (wchar_t *)

(Parentheses inside show a regular value style name.)

[Attention]
The character in the character constant ('..' style) must not be multi-byte character.
Namely, 'Kan' is not possible description.(It will become warning.)

A.6.2.1 Multi-byte character

This is the one that expressed 1 character of Japanese-Kanji with 1 or more plural byte.

- * And this is expressed with a character string (the array type of char).
- * Even if it is a simple only 1 character, it becomes the form of a character string as "字".
- * Length of the multi-byte character is not stable. Therefore, if you needs to searching, insertion and elimination for a multi-byte character in a multi-byte string (continuation of the multi-byte characters), it is necessary to confirm continually boundaries between neighbor characters during the processing.
- * On the other side, there is the advantage that conventional functions (the standard function of printf and strcpy etc.) that handle the byte string can be used without changing it.

A.6.2.2 Wide character

This expresses one character of Japanese-Kanji.

- * The wide character can be declared with the wchar_t type. This type is defined with stddef.h standard header.
- * The wide character string (the array of the wide characters) has so many data volume in comparison with the multi-byte string.
- * On the other side, because one element becomes one character, there is the advantage that wide character string can be searched, inserted and eliminated at an optional position without concerning the gap between the characters.

[Attention]
In CC32R, the wchar_t type is equal to the signed short type.
(However, do not suppose it in your programming.)

A.6.3 Programming that used a Japanese processing function

This is an example of the programming that used a Japanese processing function.

```
#include <stddef.h>      /* Wchar_t type definition  */
#include <locale.h>      /* Due to the control of locale  */
#include <stdlib.h>      /* Due to the mbstowcs function application  */
#include <stdio.h>
#include <string.h>

char     str1[] = "CC32R";
char     str2[] = "漢字 ? 数文字 "; /* (1) Description of Japanese with
                                          a multi-byte character string */
wchar_t wstr[] = L":文字列 ";      /* (2) Description of Japanese with
                                               a wide character string */

#define BUFSIZE  256
#define WBUFSIZE 128

char     buff[BUFSIZE];
wchar_t wbuff[WBUFSIZE];


void     kanjiout(wchar_t wc)     /* The function for outputting wide
                                                  character (dummy) */
{
         /* Make program as that wc is outputted to the display devices */
         return;
}

int
main(void)
{
         int     size_wch, i;

         /* (3) Multi-byte character string if there be not division etc.
                      conventional character string processing possibility */
         strcpy(buff, str1);
         strcat(buff, str2);

         /* (4) Setting up the character code for preparation to use the
                                                     mbstowcs function */
         setlocale(LC_CTYPE, "sjis");/*(In the case of Shift JIS)*/

         /* (5) Transformation to the wide character string */
         size_wch = mbstowcs(wbuff, buff, WBUFSIZE);

         /* (6) Sending each 1 character to output function while taking out
                each 1 letter from wide character string that transformed */
         /* (Yet, "?" and "No" will be omitted) */
         for (i = 0; i < size_wch; ++i) {
                 if (wbuff[i] != L'?' && wbuff[i] != L'数 ')
                         kanjiout(wbuff[i]);
         }

         /* (7) Sending 1 character to output function while taking out each
                             1 letter from another wide character string */
         for (i = 0; i < sizeof(wstr); ++i) {
                 kanjiout(wstr[i]);
         }

         return 0;
}
```

A.8 **Programming that used a Japanese processing function**

This program processes the character string including Japanese that was written in the str1, str2, wstr, and sends each 1 character to output function.
Although the kanjiout function of this example does not work yet, this program can be applied to the usage such as the likes that carries out the output to the indicator.
It explains about each part of the program below.

(1) Description of Japanese with a multi-byte character string

```
char str2[] = "漢字 ? 数文字 ";
```

String can be described similarly as conventional character string letteral, even if it includes Japanese characters.

(2) Description of Japanese with the wide character string

```
wchar_t     wstr[] = L": 文字列 ';
```

'L' is attached to the top of character string letteral in the case of a wide character string. Even both the Japanese character and conventional characters can be described in wide string.

(3) Copy and connect of a multi-byte character string

```
strcpy(buff, str1);
strcat(buff, str2);
```

As for the multi-byte character string the processing such as the copy and connection are possible like the conventional character string.

(4) Setting of a character code

```
setlocale(LC_CTYPE, "sjis");
```

Setting up the locale name (the character code name of Table A.3) by the setlocale function for converting from multi-byte character string to wide character string in the next part(5). At this time, the same character code as M32RKOUT needs to be designated, to LC_CTYPE.
If the M32RKOUT is undefined, please designate "sjis" (for PC version) or "euc" (for EWS version) to the setlocale function.

(5) Transformation to a wide character string

```
size_wch = mbstowcs(wbuff, buff, WBUFSIZE);
```

Transforming from the multi-byte character string that made it in the above (3) to the wide character string for the byte unit processing(in next (6)).

(6) Taking out each a character and outputting it (except particular character).

```
for (i = 0; i < size_wch; ++i) {
     if (wbuff[i] != L'?' && wbuff[i] != L' 数 ')
    kanjiout(wbuff[i]);

}
```

The wide character string is a wchar_t type array.
Taking out each a character, removing one of the particular characters (those are '?' or 'No' in this program) of it, and outputting it to the function.

(7) Taking out each a character from another string and outputting it

```
for (i = 0; i < sizeof(wstr); ++i) {
   kanjiout(wstr[i]);

}
```

It is similar to above (6), taking out each a character, ouputting it to the function.
This is a simple case that letter check is nonexistent.

A.6.4 Restriction items

A.6.4.1 Message display

When the compiler output the Japanese-Kanji string in the compile message, it may be octal number instead of original character.

Example) A warning message to '漢 ' (M32RKOUT a case of sjis)

warning: \0212\0277: unportable character constant

(\0212\0277 is 2 octal notation of Kan (0x8a and 0xbf) in sjis.)
The display result is different each in host environments.

A.6.4.2 Attention on multi-byte character string processing

Because the boundary between the characters is unstable, if you need process a mult-byte character string as conventional char type (argument), please carry out several operations (show below) only in that this boundary is known clearly.
Otherwise, a character will be broken or searching will finish incorrectly.

* A character string is divided in an optional place.
* A character and character string are inserted in an optional place.
* The forwarding of a character string is discontinued halfway.
* A character is searched from an optional place.
* A string is compared with other string from an optional place.

A.6.4.3 Attention on standard function use

The following functions are supposing always that LC_CTYPE is "C" locale.
Therefore, pay attention in the case that the string or character including the Japanese character is processed with these functions.

(1) Character string operation function (Function that belongs to string.h)

Because this function does not know boundary between multi-byte characters, if a multi-byte character has Japanese-Kanji character, the function does not act along to expectation.

* memchr, strchr, strpbrk, strrchr functions ... The searching may not act normally.
* strcspn, strspn functions ... The exact character number may not be returned.

(2) Character operation function (Function that belongs to ctype.h)

This function can not process it normally, in the case that the Japanese character is designated.

Example) Do not call a function as follows.

```
isprint(L'漢')
isupper(L'A')     /* Zenkaku A */
```

A.6.4.4 Correspondence of the assembler

Japanese is unable to be described with assembly language.

A.6.4.5 Correspondence of the relation tools

If the debugger that is for CC32R V.3.00 or older or is not corresponding to the Japanese character displays the Japanese character may not be displayed correctly.

A.6.4.6 preprocessor output (-P,-E option)

The compiler that the -P or -E option is designated outputs always the Japanese letter with euc.

A.6.4.7 Restriction item of Unicode

Pay attention to the next point, in the case of programming to use utf8 (Unicode).

(1) Character that is able to use it

Only the character in the 6.1(1) can be used even if environment variable is designated to utf8.
The program including the character other than Japanese that is defined with JIS X 0201/ 0208 is not able to process normally.

(2) Character that differs by the implement

The number that is assigning the Unicode correspondence to the same character by implement of host environment (an editor) and target environment, even if it is called may differ.
Because of this, if utf8 is designated to environment variable M32RKIN, M32RKOUT, please use and pay attention to the handling of Unicode.

(3) Handling of a resemblance character

The character that was inputed will be transforming to euc at first in the compiler inside. The several of the character that form resembled on Unicode are normalized to same character on euc.Therefore, even if both M32RKIN and M32RKOUT are utf8, several characters of source and output may not be equal.

A.6.5 The supplement of Japanese processing

A.6.5.1 Inside expression of the Japanese character

(1) Inside expression of the multi-byte character string

The multi-byte character string on C source will be transformed to the character code that is designated by M32RKOUT, and be transformed to the text stream (image of the content of the text file) of 8bit.

Example) Multi-byte character string

Next some program is shown description of the same meaning as
char kanji1[] = "漢字1";.

```
/* sjis  */ char kanji1[] = {0x8a,0xbf,0x8e,0x9a,0x31,0};
/* euc   */ char kanji1[] = {0xb4,0xc1,0xbb,0xfa,0x31,0};
/* utf8  */ char kanji1[] = {0xe6,0xbc,0xa2,0xe5,0xad,0x92,0x31,0};
```

(2) Expression of a wide character

- * It is euc or sjis, if the multi-byte character is transformed a wide character, the highest byte of wide character is 1st byte of multi-byte character, the lowest byte of wide character is 2nd byte of multi-byte character.
- * In same environment, if the 1byte multi-byte character is transformed a wide character, the highest bytes of wide character is zero, the lowest byte of wide character is multi-byte character.
- * It is utf8, if the multi-byte character is transformed a wide character, this value is UTF-16.

Example 1) wide character string

Next some program is shown description of the same meaning as
wchar_t kanji2[] = L"漢字2"; .

```
/* sjis  */ wchar_t kanji2[] = {0x8abf,0x8e9a,0x0032,0x0000};
/* euc   */ wchar_t kanji2[] = {0xb4c1,0xbbfa,0x0032,0x0000};
/* utf8  */ wchar_t kanji2[] = {0x6f22,0x5b57,0x0032,0x0000};
```

Example 2) wide character

Next some program is shown description of the same meaning as
wchar_t kanji3 = L'漢';

```
/* sjis  */ wchar_t kanji3 = 0x8abf;
/* euc   */ wchar_t kanji3 = 0xb4c1;
/* utf8  */ wchar_t kanji3 = 0x6f22;
```

char kanji4 = L'漢'; It becomes an overflow.

A.6.5.2 Standard library

(1) Designate the locale (locale.h)

It is possible designated a character code name that shows it in Table 9 in addition to locale, conventional "C" a locale name as it is.

(2) **A wide character transformation function (mbtowc, mbstowcs, mblen, wctomb, wcstombs)**

LC_CTYPE processes on the basis of each character code in the case of euc, sjis and also utf8 locale.

(3) Function of printf, scanf series

A format designated character string considers as the multi-byte character string of a character code that was designated to LC_CTYPE.

(4) Other standard functions

The locale will be regarded that it is the same action as C locale always.

(5) About shifting condition

A standard library function is not remembering shifting condition.
It supposes that shifting condition is already initialized when it is called.

Appendix B

The C Standard Library Function List

This appendix lists the C standard library functions and their summaries by function group.

Program diagnostic function

assert	Adds a diagnostic function to a program.

Character handling function

isalnum	Judges whether a letter or decimal number.
isalpha	Judges whether a letter or not.
iscntrl	Judges whether a control character or not.
isdigit	Judges whether a decimal number or not.
isgraph	Judges whether a printable character other than a space.
islower	Judges whether a lower case letter or not.
isprint	Judges whether a printable character including a space.
ispunct	Judges whether a special character or not.
isspace	Judges whether a space or not.
isupper	Judges whether an upper case letter or not.
isxdigit	Judges whether a hexadecimal number or not.
tolower	Converts an upper case letter into lower case.
toupper	Converts a lower case letter into upper case.

Mathematics function

acos	Obtains the arc cosine of a floating-point number.
asin	Obtains the arc sine of a floating-point number.
atan	Obtains the arc tangent of a floating-point number.
atan2	Divides a floating-point number by a floating-point number and obtains the arc tangent of the result.
ceil	Computes the integer ceiling of a floating-point number.
cos	Obtains the cosine of radians of a floating-point number.
cosh	Obtains hyperbolic cosine of a floating-point number.
exp	Obtains the exponential function of a floating-point number.
fabs	Obtains the absolute value of a floating-point number.
floor	Cuts off the fraction of a floating-point number.
fmod	Multiplies fractions of two floating-point numbers and obtains the remainder.
frexp	Divides a floating-point number into products of value (0.5, 1.0) and 2 to the *n*th power.
ldexp	Performs multiplication of a floating-point number and 2 to the *n*th power.
log	Obtains natural logarithm of a floating-point number.
log10	Obtains the base 10 logarithm of a floating-point number.
modf	Divides a floating-point number into integer and decimal.
pow	Obtains a floating-point number to *n*th power.
sin	Obtains the sine of the radians of a floating-point number.
sinh	Obtains hyperbolic sine of a floating-point number.
sqrt	Obtains the positive square root of a floating-point number.
tan	Obtains the tangent of the radians of a floating-point number.
tanh	Obtains hyperbolic tangent of a floating-point number.

Non-local jump function

longjmp	Recovers the execution environment saved by setjmp and transfers control to the program location of a setjmp call.
setjmp	Saves the current environment to a memory area.

Variable arguments access function

va_arg (Macro)	Gets variable arguments in turn.
va_end (Macro)	Ends the reference to variable arguments.
va_start (Macro)	Initializes to reference variable arguments.

Input/output function

clearerr	Clears an error condition in a stream.
fclose	Closes a file.
feof	Checks if the end of a stream is reached.
ferror	Checks if a stream is in an error condition.
fflush	Outputs the contents of a stream to a file.
fgetc	Gets a character from a stream.
fgetpos	Locates the current position on a stream.
fgets	Gets a string from an input stream.
fopen	Opens a file.
fprintf	Outputs data to a stream, according to the format.
fputc	Outputs a character to a stream.
fputs	Outputs a string to a stream.
fread	Transfers data from a stream to a memory area.
freopen	Closes a currently opened stream, and reopens a new file with the new file name.
fscanf	Gets data from a stream, and converts the data by following the format.
fseek	Moves the current read/write position within a stream.

fsetpos	Changes the current position on a stream.
ftell	Locates the current read/write position in a stream .
fwrite	Transfers data from a memory area to a stream.
getc	Gets one line from a stream.
getchar	Gets a character from the standard input file (stdio).
gets	Gets a string from the standard input file (stdio).
perror	Outputs the error message corresponding to the error code to the standard error file (stderr).
printf	Converts data by following the format and outputs it to the standard output file (stdout).
putc	Outputs a character to a stream.
putchar	Outputs a character to the standard output file (stdout).
puts	Outputs a string to the standard output file (stdout).
remove	Deletes a file.
rename	Renames a file.
rewind	Moves the current read/write position on a stream to the beginning of the file.
scanf	Gets data from the standard input file (stdin) and converts the data by following the format.
setbuf	Defines a buffer for an I/O stream.
setvbuf	Defines and sets a buffer for an I/O stream.
sprintf	Converts the data by following the format and outputs the data to an area.
sscanf	Gets data from a memory area and converts the data by following the format.
tmpfile	Creates a temporary file.
tmpnam	Creates a named temporary file.
ungetc	Returns a character a stream.
vfprintf	Outputs a variable argument list to a stream by following the format.

vprintf Outputs a variable argument list to the standard output (stdout) by following the format.

vsprintf Outputs a variable arguments list to a memory area by following the format.

General utility function

abort Puts the running program to forced stop.

abs Obtains the absolute value of an int type integer.

atexit Catalogs the function to be called upon successful termination of the program.

atof Converts the character string representing a number into a double type floating-point number.

atoi Converts the character string representing a decimal number into a int type integer.

atol Converts the character string representing a decimal number into a long type integer.

bsearch Performs binary search.

calloc Allocates a memory space and initializes the allocated memory space to 0.

div Divides an int type integer and obtains the quotient and remainder.

exit Terminates the program.

free Releases the specified memory area.

getenv Gets the content of an environmental variable.

labs Obtains the absolute value of a long type integer.

ldiv Divides a long type integer and obtains the quotient and remainder.

malloc Allocates memory area.

mblen Obtains the number of bytes composed of multibyte characters.

mbstowcs Converts a multibyte character string into a wide character string.

mbtowc Converts a multi byte character into a wide character.

qsort	Performs sorting.
rand	Generates a pseudo-random integer which resides between 0 and RAND_MAX.
realloc	Changes the memory area size to the specified size.
srand	Sets the initial value of the pseudo-random integer which the rand function generates.
strtod	Converts a string representing a number into a double type floating-point number.
strtol	Converts a string into a long type integer.
strtoul	Converts a string into an unsigned long type integer.
system	Passes a command string to the host environment.
wcstombs	Converts a wide character string into a multibyte character string.
wctomb	Converts a wide character into a multibyte character.

String handling function

memchr	Locates, in a memory area, the position where a character first appears.
memcmp	Compares the contents of two memory areas.
memcpy	Copies the contents of a memory area to the destination memory area.
memmove	Moves the contents from a memory area to the destination memory area.
memset	Copies a character into the first n characters in memory area.
strcat	Links a string to the end of a string.
strchr	Locates, in a string, the position where a character first appears.
strcmp	Compares two strings.
strcoll	Compares the two strings based on the current locale.
strcpy	Copies the contents (including null characters) of the source string to the target memory area.
strcspn	Computes the length of initial segment of a string which consists of unspecified characters.

strerror	Returns the error message.
strlen	Measures the size of string.
strncat	Links the specified number of characters to the string.
strncmp	Compares specified number of characters of two strings.
strncpy	Copies the specified number of characters from the string to memory.
strpbrk	Locates the position where one of the specified characters first appears in a string.
strrchr	Locates the position where a character last appears in a string.
strspn	Computes the length of initial segment of a string which consists of specified characters.
strstr	Finds the first occurrence point of a string within another.
strtok	Separates a string into tokens.
strxfrm	Converts the string based on the current locale.

Localization function

localeconv	Initialize struct lconv.
setlocale	Sets and searches locale information.

Date and time function

asctime	Converts data and time (a struct tm) into the equivalent text string.
clock	Gets the elapsed processor time.
ctime	Converts the calendar time (a time_t value) into the equivalent text string.
difftime	Computes the difference between the two specified times.
gmtime	Converts calendar time to Coordinated Universal Time (UTC).
localtime	Converts current calendar time to the local time.
mktime	Converts date and time (a struct tm) to the calendar time.
strftime	Converts date and time (a struct tm) to the format specified.
time	Reads the current calendar time.

Signal handling function

raise	Send a signal to the executing program.
signal	Sets up a signal handler that responds to the signal.

Initialization function (non-ANSI)

_init_atexit	Initializing for atexit().
_init_base_year	Initializing for the date and time group.
_init_exit_environ	Initializing for exit().
_init_mem	Initializing for I/O, general utility, and localization groups.
_init_stdio	Initializing for I/O group.

Termination function (non-ANSI)

_action_atexit	Performs user-registered terminations.
_exit_mem	Terminating for I/O, general utility, and localization groups.
_exit_stdio	Terminating for I/O group.
_get_exit_code	Gets the exit status from exit().

Special floating-point values judgement function (non-ANSI)

These are the functions that judges whether or not the value corresponds to any special floating-point values (float or double) one of 0.0 (Zero), Infinity or NaN (Not a Number).

The floating-point operation program can use these functions, for checking previously input value of the operation, and for checking whether or not the result of the operation is special value.

Table B.1 Special floating-point values judgement function

The floating-point value type that these functions check.		Checking functions (C language formats of prototype declarations)	
		double type	float type
Infinity		int isinf(double)	int isinff(float)
Zero	0.0	int iszero(double)	int iszerof(float)
Not a Number	NaN	int isnan(double)	int isnanf(float)

These functions return 1 if input value was corresponds to the special value of each function. Otherwise, these functions return 0.
Before the judgement functions can be used, one of the following header files must be included. Choose either one that suits to your need.

mathf.h	When you use only the float type judgement functions.
math.h	When you use the float or double type judgement functions.

Note that because the header math.h includes the functions of **mathf.h**, you do not need to include **mathf.h** if you already have math.h included.

Example : A function returns 1 if input is infinity or NaN.

```
#include <mathf.h>   /* <math.h> also acceptable */
float
func(float fin)
{
  if (isinff(fin))
    return -1.0f;  /* In the case of infinity */
  if (isnanf(fin))
    return -1.0f;  /* In the case of NaN */
  return fin * 2.0f;  /* Other cases */
}
```

Appendix C

Restrictions on Usage

There are restrictions of the CC32R.
For other precautions of only this version, see the 'Precautions on using' of the next chapter.

■ How to get files that is not included the debug-informatio

C compiler cc32R, assembler as32R and linker lnk32R have come to be generating the debugging information always. Namely, the object module and load module files that these tools generate always include the debugging information.

Such a outputting debug-information is not possible to impede in those options.

The strip32R can process even the object module that compiler and assembler generated in addition to the load module that the linker. The strip32R can process even the object module that compiler and assembler generated in addition to the load module that generated the linker. In other words, if each output files are processed with strip32R after cc32R, as32R or lnk32R, these tools act as conventional CC32R (V.4.10 or before).

Example of using strip32R: (% expresses a prompt)

Usually usage:

The strip32R is able to apply to each output file of the cc32R, as32R and lnk32R. Strip32R is able to process both files of object-module (before the link) and load module (after the link).

```
% cc32R -c -o sample1.mo sample1.c
% strip32R sample1.mo
% as32R sample2.ms
% strip32R sample2.mo
% lnk32R -o sample.abs sample1.mo sample2.mo
% strip32R sample.abs
```

To process two or more files at a time:

For example, after all the compiling and the assembling completed, the strip32R can process all the files of them.

```
% cc32R -c sample1.c sample2.c sample3.c
% cc32R -c sample4.c
% as32R -c sample5.ms
% strip32R sample1.mo sample2.mo sample3.mo sample4.mo sample5.mo
```

Even the wild card can be designated.

```
% strip32R *.mo
```

■ Cautions on using the base register function with standard library for C

[The supplement of attention on using the base register function]

Combinations of the object file as follows are not recommended. (For more details, refer to the "A.1.6 Base Register Function Limitations" of the M3T-CC32R User's Manual <C Compiler>.)

(1) The combination of object files that was created in using base register function and in not using this function.

(2) The combination of object files that was created by using different access control files.

[Attention to use the base register function and C standard library in same time]

Attached C standard library was created when the base register function is ineffective. Therefore, attached C standard library and the object file that used the base register function correspond to above (1).
In such case, the base register does not have the base address when the standard library function is executing. The base register will returns the base address after these standard functions, although the base register will not have the base address when as follows:

(1) Interrupt processing routine
Because the interrupt process happens during execution of standard library functions, you must think value of the base register is undefined.

(2) User function that is called from the particular standard library functions (qsort, bsearch etc.)

[Solutions]

When the base register function and the C standard library are used in same time, please use one of the solution methods following (1) and (2).

(1) Create a special standard library by using same access control file from the user program. And replace present standard library with it.

(2) Re-compile interrupt processing routine and user function that is called from the partcular standard library functions (qsort, bsearch etc.) by not using the base register function.

■ Avoiding the integral zero-division problem of M32R/ECU series

In M32R/ECU Series Microcomputer, if zero division calculation (its divisor is equal zero) is executed for integral division instructions (they are DIV, DIVU, REM and REMU. abbreviated as DIV-instructions), the result will be inaccurate calculations for some instructions that are executed immediately after 0 division.

For more details, refer to the Technical News No.M32R-06-0301 “M32R/ECU series Usage Notes for 0 Division Instruction”.

The correspondence in CC32R and explain about avoiding the zero-division problem by -zdiv option below.

[Correspondence methods]

The case of C language program or assembly language program

(1) Please re-program so the zero-division does not occur in logical, following the tehnical news suggests. CC32R generates the DIV-instructions to the integral calculations both divisions (/ and also /=) and remainders (% and also %=) of C language, please program so that the divisor do not become 0. Also, in assembly language, please program so that the second parameter of DIV-instructions (it means divisor) do not become 0.

(2) If you can not accomplish (1) completely, re-compile or re-assemble with -zdiv option insted of (1).

The case of using standard libraries

Even if the DIV-instructions computes the zero-dividion in the standard library functons, the problem does not occur. It is because the standard library is already treated about avoiding this problem.
Furthermore, The functions of the zero-division measurement libraries (m32RcRZ.lib, m32RcRZM.lib, m32RcRZL.lib) that was prepared in CC32R V.4.10 Release 1, have been incorporated to general standard libraries (m32RcR.lib, m32RcRM.lib, m32RcRL.lib). Because of this, If you have been using CC32R V.4.10 Release 1 and use the zero-division measurement libraries, please use general standard libraries instead of them.

The case of using non-standard libraries

In use the customer-made libraries or the re-build libraries from the standard library sources set of attachment to CC32R, please re-build or re-compile with -zdiv option.

[Explanation of the -zdiv option]

When it uses in compiling with cc32R

Compiling with -zdiv option, it generates assembly source with inserting NOP instructions each after the all of created DIV-instructions. Also, it inserts NOP instructions as same in asm functions too.
However, if you use -zdiv option with -S or -CS in same time, compiler generates assembly source with removing comment and coverting alphabetic letters to upper.
In the case of inputting assembly sources to cc32R, it performs same from assembling by as32R.

When it uses in assembling with as32R

If assemble code includes DIV-instructions with -zdiv option, it inserts NOP instructions each after the all of this DIV-instructions. However, it except case of that NOP instruction already exists after the DIV-instruction.
It means there is not following object between the DIV-instruction and the NOP-instruction. In other words, the compiler inserts NOP instruction after the DIV-instruction, if there is following object between the DIV-instruction and the NOP-instruction.

(1) Labels

(2) Generic M32R instructions except NOP instruction

(3) as32R pseudo-instructions influencing the code areas (as follows)
.ALIGN .DATA .DATAB .END .FDATA .FDATAB .FRES .RES
.SDATA .SDATAB .SECTION

On indirect calling a function that has variable arguments

The program will not run correctly if a function having a variable argument is called indirectly by using a pointer variable to a function without prototype declaration.

[Code Example]

```
#include <stdio.h>
int (*funcptr)() = printf;
int main (void) {
    (*funcptr) ("calling printf with %d\n", 1);
}
```

[Solution]

Include a prototype declaration for the pointer variable to the function. (Rewrite the above code as follows.)

```
#include <stdio.h>
int (*funcptr) (const char *,...) = printf;
int main(void) {
    (*funcptr) ("calling printf with %d\n", 1);

}
```

Data definition within the code section

The assembler outputs a warning (warning: caution! there are some data in code section) so as to alert you to data items (or space areas) present in the code section.
It is recommended to put data items in the data section.
You can suppress this warning by use of the option "-warn_suppress_code_data".

Use of preprocessor variables inside a macro body

If, as in the following example, a preprocessor variable appears starting in the first column of the line immediately after a macro call in the macro body, the preprocessor variable may not be correctly expanded when the macro call is effected.

[Code Example]

```
    .macro INST_MACRO
    MOV        #0,R0
    .endm
    .macro LABEL_MACRO label
    INST_MACRO
\label:                    ; putting a preprocessor variable from the first column
    .endm
    .section P,code,align=2
    LABEL_MACRO L1         ; this expansion will be failed
    LABEL_MACRO L2         ; this expansion will be failed

    .end
```

[Solution]

Inside a macro body, write a preprocessor variable from the second column or the subsequent.

■ About compiling the functions of 500 or more lines

When you compile a program that has the big functions of 500 or more lines by CC32R, a error "Out of memory" will occur.
In this case, divide this function so that its lines decrease.

■ Precautions about changing C Calling Convention

CC32R V.3.00 Release 1 (or newer) always generates code for function parameters by registers. Accordingly, objects of CC32R V.3.00 Release 1 (or newer) and V.2.10 Release 1 can't be linked without measuring. Correspond in the following methods.

(1) C language program that passes the function argument by using stack

It means objects and libraries that was compiled by the CC32R V.2.10 Release 1 without -RBPP option.

[How to adapt]

Compile them with CC32R V.3.00 Release 1 (or newer).

(2) Program of the assembly language that is handing over the function argument by stack

It is the program of the assembly language passing the argument of the function by using stack, and that calls function of C language or is called from it. (They include start up program and low level library functions.)

[How to adapt]

- Change the assembly language program in accordance with the setting rule of the function argument of V.3.00 Release 1. (Refer to the chapter of "the C calling rule" of the M3T-CC32R user's manual <C Compiler>.)
- When function passes the argument by registers, this function name is not under score (_) to the top but dollar mark ($) is added in object file. You need to change the function name in the assembly language that you have this to the name that complied with.

When you links these programs (above (1) and (2)) without this adaptation and program made for CC32R V.3.00 Release 1 (or newer), the error "external symbol not defined" will occur.

M3T-CC32R V.4.30 User's Manual <C Compiler>

Rev. 1.00
September 01, 2004
REJ10J0514-0100Z

M3T-CC32R V.4.30
User’s Manual <C Compiler>

Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan