

TOSHIBA

TLCS-900 C Compiler Reference

1st Edition

TOSHIBA Corporation Semiconductor Company

RESTRICTIONS ON PRODUCT USE

- The information contained herein is subject to change without notice. (W11AE-01)
- TOSHIBA is continually working to improve the quality and reliability of its products. Nevertheless, the hardware and/or software incorporated in the TOSHIBA products listed in this document ("TOSHIBA Products") in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress. It is the responsibility of the customer, when utilizing TOSHIBA Products, to fully comply with the standards of safety in making safety design for the entire system, and to avoid the situations in which a malfunction or failure of such TOSHIBA Products could cause loss of human life, bodily injury or damage to property.
In developing your designs, please ensure that TOSHIBA Products are used within specified operating ranges as set forth in the specifications for this product, the specifications for the semiconductor devices under evaluation, and any other related information. Also, please keep in mind the precautions and conditions set forth in the "TOSHIBA Semiconductor Reliability Handbook" and "Instruction Manual" or "Operation Manual" that accompany this product and any devices connected to this product.
Please always confirm the latest information of the TOSHIBA Products released on the web page of microcomputer in the web site of TOSHIBA Semiconductor Company.
(<http://www.semicon.toshiba.co.jp/eng/>) (W01AE-01)
- The TOSHIBA Products are intended for usage in the functional evaluation of semiconductor devices. TOSHIBA Products shall not be used for purposes other than functional evaluation, such as for verification of device reliability. The TOSHIBA Products shall not be incorporated this product into customer products. The TOSHIBA Products shall not be converted, disassembled, modified, or used outside its specified operating range of the TOSHIBA Products listed in this document.
- The TOSHIBA Products are intended for the functional evaluation of semiconductor devices that are designed for use in general electronics applications (e.g., computer, personal equipment, office equipment, measuring equipment, industrial robotics, and domestic appliances). These TOSHIBA Products are neither intended nor warranted for usage in equipment that requires extraordinarily high quality and/or reliability or a malfunction or failure of which may cause loss of human life or bodily injury ("Unintended Usage").
Without limiting the generality of the foregoing, unintended Usage include atomic energy control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, combustion control instruments, medical instruments, and all types of safety devices. The TOSHIBA Products shall not be used for Unintended Usage. (W02BE-01)
- The products described in this document shall not be used or embedded to any downstream products of which manufacture, use and/or sale are prohibited under any applicable laws and regulations. (W03AE-01)
- TOSHIBA does not take any responsibility for incidental damage (including loss of business profit, business interruption, loss of business information, and other pecuniary damage) arising out of the use or disability to use the product. (W04AE-01)
- The information contained herein is presented only as a guide for the applications of our products. No responsibility is assumed by TOSHIBA for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patents or other rights of TOSHIBA or the third parties. (W06AE-02)
- Product names mentioned herein may be trademarks of their respective companies. (W07AE-02)

Preface

Thank you for using Toshiba microcomputer products.

This manual describes how to use the microcomputer development system product you have purchased. Please keep this manual to hand when you use the product.

Toshiba will continue to make every effort to improve our products to better meet the needs of our customers. We will highly appreciate your continued patronage of Toshiba microcomputer products also in future.

- Microsoft®, Windows®, Windows® 2000, Windows® XP, and Windows Vista® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Technical support

The "readme.txt" file is included with the product package to help you use this product. If you have any further questions regarding the content of this manual, please do not hesitate to contact your local Toshiba sales representative.

Our technical support service is available if you encounter any phenomenon that seems to be faulty while using this product. At your request we will investigate the cause of the phenomenon and report back to you. To use this service, you need to provide us with the data that enables us to reproduce the phenomenon, such as the operation procedure, etc. Please note that we may not be able to deal with a phenomenon that cannot be reproduced.

INDEX

Part 1	About this book	1
Chapter 1	Explanation of this manual	3
1.1	How to Read the Manuals	3
Part 2	Introduction	5
Chapter 2	Compiler Introduction	7
2.1	Section.....	7
2.1.1	What Is a Section?.....	7
2.1.2	Section Types.....	8
2.1.3	Section Names	8
2.2	Displacement	9
2.2.1	What Is Displacement?.....	9
2.2.2	Displacement Types	9
2.2.3	Relationship Between Sections and Displacement	9
Part 3	C Language Specifications.....	11
Chapter 3	Toshiba Extended Specifications	13
3.1	Priority Sequence of Extended Specifications	13
3.2	Extended Qualifier	13
3.2.1	Function Qualifier.....	13
3.2.2	Memory Qualifier	14
3.2.3	Other Qualifier.....	15
3.3	#pragma Directives.....	15
3.3.1	Function type.....	15
3.3.2	Displacement	15
3.3.3	Extern Directive.....	16
3.3.4	I/O.....	16
3.3.5	Disable Interrupt Directive	16
3.3.6	Structure Packing Directive	17
3.3.7	Restriction of Warning Output Directive.....	17
3.3.8	PID Directive.....	18
3.4	Options.....	18
3.5	Default Status	18
3.5.1	Address and Displacement	18
3.5.2	Pointer Displacement	18
3.5.3	Section Name and Displacement	19
3.6	Alignment.....	19
3.7	Shift Operations	19
3.8	Intrinsic Functions	19
3.9	Inline Assembly	20
3.9.1	__ASM()	20
3.9.2	__asm()	20
3.10	Register Pseudo Variable	20
3.10.1	General Register Pseudo Variables.....	21
3.10.2	Control Register Pseudo Variables	21
3.11	Pre-defined Macros	22
3.12	Non-ANSI Specifications	22
3.12.1	Integral Promotion.....	22
3.12.2	Arithmetic Conversion of Multiplication and Division	23
3.12.3	Numerical Number	24
3.13	PIC Specifications.....	25

3.14	PID Specifications.....	26
Part 4	Generating Execution Programs from the Command Line.....	27
Chapter 4	Compiler Driver Overview	29
Chapter 5	Option Descriptions.....	30
	-# Display the tool that activates the link process from compiling	30
	-A Compile in accordance with ANSI specification	30
	-D Define the preprocessor macro	30
	-E Output preprocessor processing results to standard output.....	31
	-F Specify the value which fill a blank area.....	31
	-I Specify Search Path for Include Files	32
	-J Recognize the kanji code included in a source file	32
	-L Specify a search path for input file to the linker	32
	-Mi No Processing Macro Preprocessor	33
	-Nb Select CPU Type	33
	-O Perform Optimization.....	33
	-P Execute only the preprocessor	34
	-S Create assembly language source program	34
	-U Disable the specified preprocessor macro definition.....	35
	-V Output version information.....	35
	-W Transfer option to specified tool	36
	-XA, -XC, -XD Change the Section Alignment.....	36
	-XE Handles \ in assembler source files as a character.....	37
	-XF Leave an assembly language source program	37
	-XS Output the code by size optimization.....	37
	-Xaa, -Xac, -Xad Create the section for 1 byte variables	38
	-Xc Specify the function type to cdecl.....	38
	-Xec Change the type of the enumeration constant.....	39
	-Xns Deter Integration of Stack Freeing	39
	-Xp Packing the structure.....	40
	-Xr Specify the function type to cdecl	40
	-Xsc Change the processing method of shift operation.....	40
	-Xub Handle bit field members as unsigned	41
	-Xuc Handle the char type as unsigned char type.....	41
	-Xw Change the bit field allocation sequence.....	41
	-ZA, -ZC, -ZD, -ZT Change the default displacement	42
	-Za, -Zc, -Zd, -Zt, -Zi Change the section name and the displacement.....	42
	-c Create a relocatable object.....	43
	-e Create an error list file.....	43
	-f Read the file that describes the option to be used.....	43
	-g Outputs debugging information	44
	-l Create a list file	44
	-o Specify an output file name	45
	-r Perform incremental linking	45
	-s Perform Macro Preprocessor Identifier definition.....	46
	-u Ignore predefined macros	46
	-u Register undefined symbols to the symbol table.....	46
	-w Set the Warning Level.....	47
Part 5	Optimizations.....	49
Chapter 6	Optimizations.....	51
6.1	Optimizations by C compiler	51
6.1.1	Assignment of Variables to Registers.....	51
6.1.2	Integrated Stack Release	51
6.1.3	Minimum Optimization	52
6.1.4	Branch Optimizations	52
6.1.5	Unnecessary Instructions Elimination	52
6.1.6	Copy Propagation	53

6.1.7	Common Sub-expression Elimination	53
6.1.8	Loop Optimizations	54
6.2	Optimizations by User	54
6.2.1	Optimizations when Describing Source Program	54
6.2.2	Optimizations by Option	54
Part 6	Standard Library Functions	57
Chapter 7	Standard Library Overview	59
Chapter 8	Header File	60
Chapter 9	Library Function	61
9.1	Standard Library Function List	61
9.2	Runtime Library List	63
Part 7	Error Message	65
Chapter 10	Error Message Format	67
10.1	Types of Error Messages	67
10.2	Error Message Format	67
Chapter 11	Driver Error Message	68
11.1	Driver Fatal Error	68
11.2	Driver Warning	68
Chapter 12	C Compiler Error Message	69
12.1	C Compiler Fatal Error	69
12.2	C Compiler Error	70
12.3	C Compiler Warning	77
APPENDIX	85
A	ANSI Processing System Dependence Specifications	87
A.1	Definitions and Conventions	88
A.1.1	[3.4 Byte]	88
A.1.2	[3.14 Object]	88
A.2	Environment	88
A.2.1	[5.1.1.2 Translation phases]	88
A.2.2	[5.1.2.1 Freestanding environment]	88
A.2.3	[5.1.2.2.1 Program startup]	89
A.2.4	[5.1.2.3 Program execution]	89
A.2.5	[5.2.1 Character sets]	89
A.2.6	[5.2.4.2.1 Sizes of integral types]	89
A.2.7	[5.2.4.2.2 Characteristics of floating types]	91
A.3	Language	93
A.3.1	[6.1.2.5 Types]	93
A.3.2	[6.1.3.1 Floating constants]	93
A.3.3	[6.1.3.4 Character constants]	93
A.3.4	[6.1.7 Header names]	94
A.3.5	[6.2.1.1 Characters and integers]	94
A.3.6	[6.2.1.2 Signed and unsigned integers]	94
A.3.7	[6.2.1.3 Floating and integral]	95
A.3.8	[6.2.1.4 Floating types]	95
A.3.9	[6.3.2.3 Structure and union members]	95
A.3.10	[6.3.3.4 The sizeof operator]	95
A.3.11	[6.3.4 Cast operators]	96
A.3.12	[6.3.5 Multiplicative operators]	96
A.3.13	[6.3.7 Bitwise shift operators]	96

A.3.14	[6.5.1 Storage-class specifiers]	96
A.3.15	[6.5.2.1 Structure and union members]	97
A.3.16	[6.5.3 Type specifiers]	97
A.3.17	[6.8.1 Conditional inclusion]	97
A.3.18	[6.8.2 Source file inclusion]	98
A.3.19	[6.8.6 Pragma directive]	99
A.3.20	[6.8.8 Predefined macro names]	99
B	Translation Limits	100

Part 1 About this book

Chapter 1 Explanation of this manual

This manual includes specifications relating to C language, compiler option contents, and the like.

1.1 How to Read the Manuals

Here, we will explain the format description rules.

Format Description Rules

[Format Description Example]

<pre>#pragma section <Section Type> [<Section Name>] [<Displacement> <Start Address>]</pre>

#pragma section For commands and options, etc., parts that do not have the enclosure symbols or delimiting symbols described hereafter are noted as is in the actual program.

<Section Type> Specifiers enclosed in < > describe character strings or numerical values specified within < > in the actual program.

[<Section Name>] Specifiers enclosed in [] can be omitted in the actual program.

[<Displacement> | <Start Address>]

For specifiers delimited by " | ", specify one of those items in the actual program.

Part 2 Introduction

Chapter 2 Compiler Introduction

This manual provides information about the specifications of the compiler contained in the TLCS-900 family C Compiler. In this chapter, we will explain the terminology used in this manual.

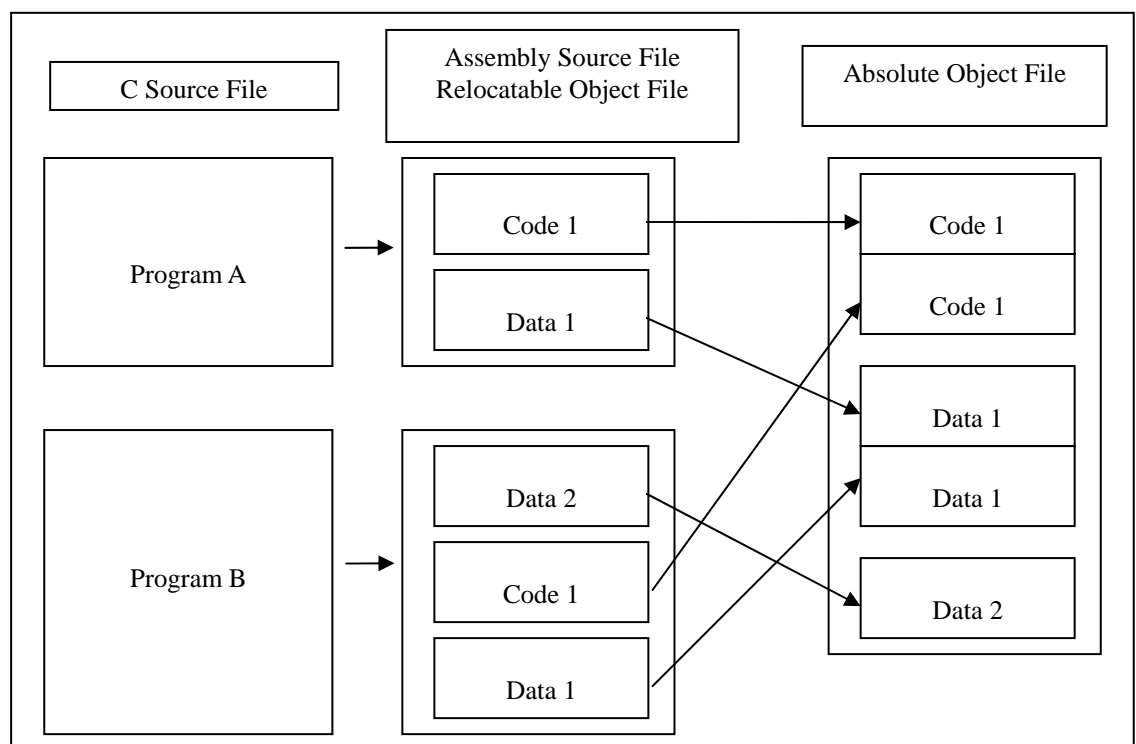
2.1 Section

2.1.1 What Is a Section?

A section indicates a respective collection of data or code categorized by type. A section is also the smallest unit when indicating memory allocation using link. Following, we will explain a summary of this.

The compiler system generates files that operate on the CPU from a program noted in C language or assembler language. These files are called absolute object files. The compiler system generates absolute object files using the following procedure.

1. The compiler translates a C language source program and generates an assembler source program, and outputs this to an assembler source file. For the output assembler source program, code and data are categorized according to type. a collection of categorized code or data is called a section.
2. The assembler analyzes the assembler source file and translates it into machine language, and generates re-allocatable files for which the address has not been determined. These files are called relocatable object files.
3. The linker allocates the final address for executing on the CPU the code or data contained in a relocatable object file. As a result, an absolute object file is generated.



2.1.2 Section Types

In this section, we will explain a detailed explanation of the section types.

With the compiler, five types of sections have been prepared, including area sections, code sections, const sections, data sections, and io sections, to express the types of code and data. Each section expresses data for which an initial specifier has not been specified, instruction code, data for which rewrite is not performed, data for which an initial specifier has been specified, and data that expresses I/O.

Table 2-1 Section Categorization

Section Name	Type	Items Subject to Categorization
area section	data	Data for which an initial specifier has not been specified
code section	code	Instruction code (mnemonic)
const section	data	Data for which rewrite is not performed
data section	data	Data for which an initial specifier has been specified
io section	I/O	Data that expresses I/O

The compiler analyzes the C source program, and depending on the data or code type, performs five type section categorization. The compiler uses section directives to define each section within the output assembler source file.

For information on section directives, see the "TLCS-900 Assembler Reference".

2.1.3 Section Names

In the five types of sections, it is possible to give a name to each item. This is called a section name.

When there is no section name specified for a C source file, a name prepared in advance by the compiler is given to each section. This name is called a pre-defined section name. Pre-defined section names are determined from section type and displacement. For information on displacement, see Section 2.2, "Displacement".

Table 2-2 Pre-Defined Section Names

Pre-Defined Section Name	Displacement	Section Type
f_area	far	area
n_area	near	
t_area	tiny	
f_data	far	data
n_data	near	
t_data	tiny	
f_const	far	const
n_const	near	
f_code	far	code
n_code	near	
io_<symbol>	Nothing (absolute address specification)	io

For information on the specification method and changing method for section names, see Chapter 3, "Toshiba Extended Specifications".

2.2 Displacement

2.2.1 What Is Displacement?

The compiler can specify to which address space on memory to allocate data such as variables or pointers. The range of this address space is called displacement.

2.2.2 Displacement Types

The compiler has displacement according to the address space range in which data is allocated. The address spaces that show each displacement are as shown in Table 2-3.

Table 2-3 Relationship Between Displacement and Address Space

Displacement	Address Space Where Data Can Be Allocated
tiny	0x0 - 0xff
near	0x0 - 0xffff
far	0x0 - 0xffffffff

The compiler attempts to output the optimal code for accessing data according to the displacement type. Because of this, there are cases when it is possible to access data with less code depending on the displacement specification.

For information on the displacement specification method, see Chapter 3, "Toshiba Extended Specifications".

2.2.3 Relationship Between Sections and Displacement

With the compiler system, data is categorized by sections. The address space which allocates data is limited according to specify the displacement to the categorized data. Therefore, the code for accessing to data is controlled.

The specification for allocation of a section on the memory is done by a file called a link command file. For information on link command files, see the "TLCS-900 Assembler Reference".

Part 3 C Language Specifications

Chapter 3 Toshiba Extended Specifications

In this chapter, we will describe the Toshiba extended specification which is a unique function of this company.

3.1 Priority Sequence of Extended Specifications

In principle, the priority sequence of related extended specifications will be according to the following.

1. Extended qualifier
2. #pragma specification
3. Option specification
4. Default status

3.2 Extended Qualifier

A extended qualifier specifies just before target symbol name.

3.2.1 Function Qualifier

A function qualifier specifies the behavior to a function.

`__interrupt`

The function specified with this function qualifier is dealt as maskable interrupt function, and "reti" is used as return statement at function exit. In this interrupt function, its return value and argument must be "void". All using registers in a function are saved at function entry and restored at exit.

`__regbank(<number>)`

The function specified with this function qualifier is dealt as maskable interrupt function, and "reti" is used as return statement at function exit. In this interrupt function, its return value and argument must be "void". A register bank is automatically changed by <number>, in this interrupt function. And the register XIX, XIY and XIZ are saved and restored.

The register bank number which is able to specifies at <number> is either of -1 to 3.

When a function for which `__regbank(-1)` is specified is called, register bank do NOT change, and register are not saved and restored.

Table 3-1 Interrupt function

	<code>__interrupt</code>	<code>__regbank(<number>)</code>	<code>__regbank(-1)</code>
how to saved / restored register	using registers in interrupt function are saved/restored	the register XIX, XIY and XIZ are saved/restored	the register do NOT saved/restored
bank switching	-	x	-

__cdecl

The function specified with this function qualifier is dealt as cdecl function. All arguments of cdecl function are passed by stack. The arguments are recognized from right side, and pushed it on the stack. The return value is stored to register "XHL". The return value which size is bigger than 4 byte is passed by stack.

__cdecl

The function specified with this function qualifier is dealt as cdecl function. All arguments of cdecl function are passed by register. The arguments are recognized from left side. However, the argument which size is bigger than 2 byte is passed by stack.

The register to be used is as follows.

- 1st argument : WA (2 byte or less), XWA (4 byte)
- 2nd argument : BC (2 byte or less), XBC (4 byte)
- 3rd argument : DE (2 byte or less), XDE (4 byte)
- 4th or after argument : Using stack. Arguments are pushed from left side.

The return value is stored to register "XHL". The return value of cdecl function is not able to specify floating point number or structure variable or union variable. Moreover, it cannot be defined as variable arguments.

__inline

The function specified with this function qualifier is dealt as inline function. The compiler embeds the actual function in the place where the __inline function is called, and reduces the function call overhead. When using inline function, because of the actual function may be embedded at two or more places, the code size may become large.

In the following case, a warning message is output and inline expansion is not performed.

- Functions defined by variable arguments
- Functions called without being defined
- Recursive functions

__pic

The function specified with this function qualifier is dealt as PIC function (Position Independent Code). For details, see Section 3.13, "PIC Specifications".

3.2.2 Memory Qualifier

A memory qualifier specifies a displacement for a variable, constant (const object), or pointer. Three types of displacement are available that __tiny, __near and __far.

Variables and constants

- __tiny
8 bit area definition (address 0x0 – 0xff)
- __near
16 bit area definition (address 0x0 – 0xffff)
- __far
24 bit area definition (address 0x0 – 0xffffffff)

__tiny cannot be specified as a constant (const object).

Pointer

The type of a pointer that stores the address value of a variable is unsigned long. The pointer itself can be allocated to tiny area, near area, or far area like a variable.

3.2.3 Other Qualifier

I/O Qualifier

I/O qualifier specifies an absolute address for which I/O are allocated.

```
<type specifier> __io(<address>) <variable name>;
```

The variables specified with this qualifier is dealt as I/O variables. An absolute address with which a I/O variables is allocated is specified as <address>, the address can be specified in the range of 0x0 to 0xfffff.

I/O variables have the same attributes as variables for which volatile is specified. An initializer cannot be specified for an I/O variable. Moreover, it cannot qualify with const to I/O variable.

PID Qualifier

PID qualifier specifies a PID(Position Independent Data) variable.

```
<type specifier> __pid <variable name>;
```

The variable specified with this qualifier is dealt as PID variable. For details, see Section 3.14, "PID Specifications".

3.3 #pragma Directives

3.3.1 Function type

#pragma cdecl

#pragma cdecl

These directives set the default function type to cdecl or cdecl. These directives are useful for function declaration and function definition.

The effective range of this #pragma directive is, until just before the next #pragma cdecl/decl directive appear from a function declaration or a function definition immediately after specifying these directives. If the next #pragma cdecl/decl directive do not appear, it is valid until the end of the file.

3.3.2 Displacement

#pragma section <section-type>

#pragma section <section-type> <section-name>

#pragma section <section-type> <section-name> <address>

#pragma section <section-type> <section-name> <displacement>

These directives change the section name for the specified `<section-type>` to `<section-name>`. You can also use `<address>/<displacement>` to specify the location in which the section will be allocated.

You cannot specify `<address>/<displacement>` simultaneously.

If you specify only `<section-type>` without specifying `<section-name>`, all displacement settings for the specified section will be reset to the default state. For `<section-type>`, specify `data`, `area`, `const`, `code`, or `io`.

If `<section-type>` is `io`, you cannot specify `<address>/<displacement>`.

You cannot use an assembler reserved word for `<section-name>`. The `<section-name>` must be a name that complies with the identifier description rules for the assembly language.

You can specify a value in the range from `0x0` to `0xffffffff` for `<address>`.

You can specify `tiny`, `near`, or `far` for `<displacement>`. However, when `<section-type>` is `const` or `code`, `tiny` cannot be specified.

The effective range of this `#pragma` directive is, until just before the next `#pragma` section directive for the same `<section-type>` appear from a function definition or a variable definition immediately after specifying these directives. If the next `#pragma` section directive for the same `<section-type>` do not appear, it is valid until the end of the file.

This directives is invalid to external variable declaration(`extern`) other than code section.

3.3.3 Extern Directive

`#pragma extern <displacement>`

This directive specifies the displacement of variable which declared external variable. You can specify `tiny`, `near`, or `far` for `<displacement>`.

The effective range of this `#pragma` directive is, until the next `#pragma extern` directive appear from an external variable definition immediately after specifying this directives.

3.3.4 I/O

`#pragma io <variable-name> <address>`

This directive specifies a variable as an I/O variable and assigns it a specified address. You must specify `<variable-name>` that is not yet defined or declared.

For `<address>`, you can specify a value in the range from `0x0` to `0xffffffff`.

3.3.5 Disable Interrupt Directive

`#pragma disinterrupt ([<level>]) <function-name> [, <function-name>...]`

This directive disables interrupts in the specified function. Specifically, it inserts a disable interrupt instruction immediately after the function call (before allocating a stack area and saving registers) and

inserts an enable interrupt instruction immediately before the function returns (after restoring registers and releasing the stack area).

For <level>, specify the interrupt level. The interrupt level is specified the number from 0 to 7. If <level> is omitted, it means that <level> is defined as 1. "(" cannot be omitted.

For <function-name>, specify the name of the function. You can also specify the name of an interrupt function besides the name of an usual function. You can specify more than one function name by delimiting them with commas in a single line. Multiple declaration of the same name function cannot be made by a different interrupt levels.

This directive must appear before the definition of the function.

3.3.6 Structure Packing Directive

#pragma pack([<alignment-byte>])

This directive specifies the alignment value of member of structure as <alignment-byte>. The alignment value can be specified the number 1 or 2. If <alignment-byte> is omitted, it means the default alignment value.

The effective range of this #pragma directive is, until just before the next #pragma pack() directive appear from a structure definition immediately after specifying these directives. If the next #pragma pack() directive do not appear, it is valid until the end of the file.

3.3.7 Restriction of Warning Output Directive

#pragma warningoff [<warning-number>, <warning-number>, ...]

#pragma endwarningoff [<warning-number>, <warning-number>, ...]

These directive restrict the output of the specified warning number.

The effective range of these #pragma directive is, until just before #pragma endwarningoff directive to the specified warning number appear from immediately after specifying #pragma warningoff directive as same number. If the #pragma endwarningoff directive do not appear, it is valid until the end of the file. If <warning-number> is omitted, it means the all warning number are specified.

These directive are valid only to warning which the parser(thc1.exe) outputs. The following warning that the code generator outputs cannot be restricted.

THC2-Warning-556 THC2-Warning-634 THC2-Warning-635
--

<Notice>

On the nature of the warning, THC1-Warning-632 which the parser outputs is outputted, after scanning to the end of a file. For this reason, in order to restrict this warning, please do not specify "#pragma endwarningoff 632" corresponding to "#pragma warningoff 632."

3.3.8 PID Directive

#pragma pid_<on | off>

This directive specifies a variable as PID(Position Independent Data). This directive valid to variable definition, and invalid to external variable declaration.

The effective range of this #pragma directive is, until just before #pragma pid_off directive appear from variable definition immediately after specifying #pragma pid_on directive. If the #pragma pid_off directive do not appear, it is valid until the end of the file. For details, see Section 3.14, "PID Specifications".

3.4 Options

The options equivalent for Toshiba Extended Specifications are as follows.

The effective range of optional feature is the whole file which specified the option, because of the options are specified to a file. For details of each option, see Chapter 5, "Option Descriptions".

Table 3-2 The Options equivalent for Toshiba Extended Specifications

Toshiba Extended Specifications	Option
Function Type	-Xc -Xr
Displacement	-Za <section-name>[, <displacement>] -Zc <section-name>[, <displacement>] -Zd <section-name>[, <displacement>] -Zt <section-name>[, <displacement>] -ZA <displacement> -ZC <displacement> -ZD <displacement> -ZT <displacement>
Structure and Union Packing	-Xp<1 2>
Change Method for Processing Shift Operation	-Xsc
Change the type of the enumeration constant	-Xec
Deter Integration of Stack Freeing	-Xns

3.5 Default Status

3.5.1 Address and Displacement

When an address is specified for an I/O variable or section, the following displacement is automatically used.

Table 3-3 Address and Displacement

Address	Displacement
0x0 - 0xff	tiny
0x100 - 0xffff	near
0x10000 - 0xffffffff	far

3.5.2 Pointer Displacement

The displacement of pointer is far. The size of pointer is unsigned 4 byte.

3.5.3 Section Name and Displacement

If you do not explicitly specify the location in which a function or variable will be allocated, it will be output to a predefined section according its type.

The following shows relationship among the type of a function or variable and the name and displacement of the predefined section to which the function or variable is output. The default sections are shaded in the table.

Table 3-4 Section Name and Displacement

Section Type	Section Name	Displacement
data	t_data	tiny
	n_data	near
	f_data	far
area	t_area	tiny
	n_area	near
	f_area	far
const	n_const	near
	f_const	far
code	n_code	near
	f_code	far
io	io_variable-name	specify by Address

3.6 Alignment

The default start alignments and allocation alignments for data, area, and const sections are 2 byte, respectively. And the default start alignments and allocation alignments for code and io sections are 1 byte, respectively.

Moreover, the stack alignment is 2 byte, an 1 byte argument is pushed on the stack after extended to 2 byte.

3.7 Shift Operations

TLCS-900 family C compiler has two kinds of processing methods of the shift operation.

The shift counter of a shift operation has constraint by default. The constraint is that low 4 bits of shift counter become effective, when shifting a variable of 16 bits or less type (char, short or int). When shifting a variable of long type, the low 5 bits of shift counter become effective.

For example, when an operation "val << 0x12" is calculated using the char type variable "val", it processes as "val << 0x02" because of low 4 bits are effective.

The shift counter is able to lose constraint by specifying -Xsc option. For details of this option, see Chapter 5, "Option Descriptions".

3.8 Intrinsic Functions

With the TLCS-900 family C compiler, the following intrinsic functions are provided.

When you use intrinsic functions, include <stdlib.h>.

Table 3-5 Intrinsic function

Intrinsic function name	Operation	Processing contents
void DI();	Issues interrupt prohibit instruction	DI
void EI();	Issues interrupt authorization instruction	EI
void EI900(<level>);	Issues interrupt authorization instruction with interrupt level (<level> is until 7 from 1)	EI <level>

<Notice>

The interrupt level is 0, when __EI() is specified.

If __DI(), __EI(), or __EI900(<level>) is described at the head of a function, DI or EI is output after the entrance processing of function(stack processing). Similarly, if __DI(), __EI(), or __EI900(<level>) is described at the end of a function, DI or EI is output before the exit processing of function(stack processing).

3.9 Inline Assembly

With the TLCS-900 family C compiler, it is possible to describe assembly language directly into a C language source file. This function is called inline assembly.

The inline assembly formats is as following.

```
__ASM(<string-constant> [,<string-constant>...]);
__asm(<string-constant> [,<string-constant>...]);
```

<Example>

```
void * word_memset(void * s, int c, size_t n)
{
    __ASM("ld    BC,(XSP+10) ; n");
    __ASM("ld    XHL,(XSP+4) ; s");
    __ASM("cp    BC,0");
    ...
}
```

3.9.1 __ASM()

The compiler performs register allocation on the assumption that all registers are saved by across inline assembly __ASM().

3.9.2 __asm()

The compiler performs register allocation on the assumption that all registers except XIZ is destroyed by across inline assembly __asm(). The HL registers is the saved register across a function call.

3.10 Register Pseudo Variable

With the TLCS-900 family C compiler, the register pseudo variable is provided to be able to directly operate the registers in the C source program. There are a general register pseudo variable and a control register pseudo variable in the register pseudo variable.

With register pseudo variables, the name has two underscores (__) added to the beginning of the actual register name, and this is allocated to the actual register. Register pseudo variable names must be in upper case letters.

The register pseudo variable has the following restrictions.

- It is not possible to use the address operator "&" for register pseudo variables.
- It is not possible to use as an argument of function.
- It is not possible to do substitute processing to global register pseudo variables.
- __SF, __ZF, and __VF are possible to use only comparison with an integer constant.
- __CF is possible to use the following processing.
 - Comparison with an integer constant.
 - Assignment integer constant (0 or 1) to __CF.
 - Assignment __CF to a bit field of 1-bit width.
- The control register pseudo variables are possible to use only the operand of simple assignment operator (contain the left operand) or comparison operator. At that case, it is necessary to typecast to each type of control register pseudo variable.

<Notice>

When using register pseudo variables, make sure that register pseudo variables do not interfere with the register which compiler uses in the middle of evaluation of expression. When using register pseudo variables, avoid using it with a complicated arithmetic expression, and surely check the register value in the assembly source files which output with -XF option.

3.10.1 General Register Pseudo Variables

The general register pseudo variables are as follows.

Table 3-6 General Register Pseudo Variable

Register Pseudo Variable Name	Type	Contents
__XWA, __XBC, __XDE, __XHL, __XIX, __XIY, __XIZ, __XSP	unsigned long	32 bit register
__WA, __BC, __DE, __HL, __IX, __IY, __IZ, __SP	unsigned int	16 bit register
__W, __A, __B, __C, __D, __E, __H, __L	unsigned char	8 bit register
__SF, __ZF, __VF, __CF	unsigned int	flag register

3.10.2 Control Register Pseudo Variables

There are Micro DMA control register, normal stack pointer, and interrupt nesting counter in the control register pseudo variables. The usable registers vary according to CPU types. For details are as following tables.

Table 3-7 Control Register Pseudo Variable for TLCS-900 series(-Nb0)

Register Pseudo Variable Name	Type	Contents
__DMAS0, __DMAS1, __DMAS2, __DMAS3, __DMAD0, __DMAD1, __DMAD2, __DMAD3	unsigned long	micro DMA control register
__DMAC0, __DMAC1, __DMAC2, __DMAC3	unsigned int	
__DMAM0, __DMAM1, __DMAM2, __DMAM3	unsigned char	
__NSP	unsigned int	normal stack pointer
__XNSP	unsigned long	

**Table 3-8 Control Register Pseudo Variable
for TLCS-900/L series and TLCS-900/L1 series(-Nb1)**

Register Pseudo Variable Name	Type	Contents
__DMAS0, __DMAS1, __DMAS2, __DMAS3, __DMAD0, __DMAD1, __DMAD2, __DMAD3	unsigned long	micro DMA control register
__DMAC0, __DMAC1, __DMAC2, __DMAC3	unsigned int	
__DMAM0, __DMAM1, __DMAM2, __DMAM3	unsigned char	
__INTNEST	unsigned int	interrupt nesting counter

Table 3-9 Control Register Pseudo Variable for TLCS-900/H series(-Nb2)

Register Pseudo Variable Name	Type	Contents
__DMAS0, __DMAS1, __DMAS2, __DMAS3, __DMAD0, __DMAD1, __DMAD2, __DMAD3	unsigned long	micro DMA control register
__DMAC0, __DMAC1, __DMAC2, __DMAC3	unsigned int	
__DMAM0, __DMAM1, __DMAM2, __DMAM3	unsigned char	
__INTNEST	unsigned int	interrupt nesting counter

Table 3-10 Control Register Pseudo Variable for TLCS-900/H1 series(-Nb3)

Register Pseudo Variable Name	Type	Contents
__DMAS0, __DMAS1, __DMAS2, __DMAS3, __DMAS4, __DMAS5, __DMAS6, __DMAS7, __DMAD0, __DMAD1, __DMAD2, __DMAD3 __DMAD4, __DMAD5, __DMAD6, __DMAD7	unsigned long	micro DMA control register
__DMAC0, __DMAC1, __DMAC2, __DMAC3 __DMAC4, __DMAC5, __DMAC6, __DMAC7	unsigned int	
__DMAM0, __DMAM1, __DMAM2, __DMAM3 __DMAM4, __DMAM5, __DMAM6, __DMAM7	unsigned char	
__INTNEST	unsigned int	interrupt nesting counter

3.11 Pre-defined Macros

Pre-defined macros added uniquely to the TLCS-900 family C compiler are as follows. These macros are defined at a value when the conditions are established. When the conditions are not established, these macros are not defined.

Table 3-11 List of Pre-defined Macros

Macro name	Condition
__TOSHIBA__	TOSHIBA Compiler
__900__	TLCS-900 family C compiler

3.12 Non-ANSI Specifications

TLCS-900 family C compiler outputs object code which conform to ANSI, when -A option specified.

If -A option is not specified, compiler utilizes the machine instruction effectively and outputs object code with sufficient object efficiency.

3.12.1 Integral Promotion

In ANSI standard, there is a rule called "integral promotion". When the integral type which is smaller than int type is used in an expression, the type convert to int type no explicitly shown in this rule. On the other hand, TLCS-900 family C compiler carry out an operation using with 8 bit registers, no conversion of integral promotion. It realizes making code size small.

If you want to operate with integral promotion in ANSI standard, -A option specified.

<Example>

```

extern int      intval1;
extern char     charval2, charval3;

void func( )
{
    intval1 = charval2 + charval3;
}

/* The case of -A option is not specify
   public _func
   _func:
       ld  A,(_charval2)    ; no conversion of integral promotion
       add A,(_charval3)    ; no conversion of integral promotion,
                           ; direct expression
       exts    WA
       ld  (_intval1),WA
       ret
   */

/* The case of -A option is specify (ANSI standard)
   public _func
   _func:
       ld  C,(_charval3)
       exts    BC           ; convert to int type
       ld  A,(_charval2)
       exts    WA           ; convert to int type
       add WA,BC
       ld  (_intval1),WA
       ret
   */

```

<Caution>

When -A option is not specify, addition of char type output result of char type, after that, the result of char type is converted to int type and assign to a result. Because of this, when the result value is out of range of char type, it does not get an expected result. In such a case, please typecast the type.

3.12.2 Arithmetic Conversion of Multiplication and Division

In ANSI standard, operands in different types are converted to the largest size in an expression. It is also applied to multiplication and division. On the other hand, TLCS-900 family C compiler calculates without carrying out a type conversion by utilizing of the multiplication and division machine instruction. Therefore, code size can be made small.

Specify -A option to process the arithmetic conversion of multiplication and division by ANSI conformity.

<Example 1>

```

extern long     longval1;
extern int      intval2, intval3;

void func( )
{
    longval1 = intval2 * intval3;
}

/* When -A option is not specified
   public _func
   _func:
       ld      WA,(_intval2)

```

```
    muls    XWA,(_intval3)
    ld      (_longval1),XWA ; direct assignment,
                                ; don't process arithmetic conversion
    ret
*/

/* When -A option is specified (ANSI standard)
   public _func
   _func:
       ld      WA,(_intval2)
       muls    XWA,(_intval3)
       exts    XWA          ; the arithmetic conversion
       ld      (_longval1),XWA
       ret
*/
```

<Caution>

When -A option is not specify, the result value assign to a long type variable without carrying out the arithmetic conversion. Because of this, when the result value is out of range of int type, it does not get an expected result.

<Example 2>

```
unsigned long  val1;
unsigned short val2;
unsigned short val3;

void func2()
{
    val1 = 0x70000000;
    val2 = 0x200;
    val3 = val1 % val2;
    /* when -A is not specified, val3 becomes indeterminate */
}

/* When -A option is not specified
   _func2:
       ld      XWA,0x70000000
       ld      (_val1),XWA
       ldw     (_val2),0x200
       ld      XWA,(_val1)
       div     XWA,(_val2)
       ; overflow occurs, XWA becomes indeterminate value
       ld      WA,QWA
       ld      (_val3),WA
       ret
*/
```

<Caution>

When -A option is not specify, mnemonic "div" may be used for division and remainder, in order to make code size small.

Because of this, it does not get an expected result, when a quotient overflows by division is performed.

3.12.3 Numerical Number

In ANSI standard, the numerical values which can be treated as an int type (the range of -32768 to 32767) are treated as the signed int type. On the other hand, in TLCS-900 family C compiler, the numerical values which can be treated as a signed char type are treated as the signed char type. It is also applied to unsigned char type.

Specify -A option to treat the value of the range of -32768 to 32767 as the signed int type.

<Caution>

When -A option is not specify, the value of the range of 128 to 255 are treated as a unsigned char type. Because of this, when these numerical values invert, it does not get an result of ANSI standard.

<code>~128 = 0x7f</code>

Specify -A option or typecast signed int type to get the value "`~128 = 0xff7f`".

3.13 PIC Specifications

PIC is the abbreviation for Position Independent Code, it is the code which is able to execute at arbitrary location.

Register for PIC

When using PIC function, "XIX" register is used as the base register. Therefore, "XIX" register can not be used for any uses other than the base register.

A initial location address of PIC is stored in "XIX" register.

Section Name

The section name in which PIC function is stored is "pic_code".

Table 3-12 Section Name of PIC Function

Section name	Target
pic_code	PIC Function

Moving PIC Function

PIC function can be moved on memory while the program is executing, and it can also execute at destination as it is. When moving PIC function, whole "pic_code" section must be moved, and the program which moves the "pic_code" section is needed.

PIC Function Pointer

Since the offset value is held, PIC function pointer can be moved on memory while the program is executing.

In this case, in order to realize NULL pointer, it is necessary to set up "pic_base" section.

"pic_base" section specifies as code section of 1 byte or more. Moreover, "pic_base" section must be specified by a link command file so that it may allocate just before "pic_code" section.

The pointer to PIC function can be initialized only by a constant or an address of PIC function. And operation of PIC function pointer and normal function pointer is not made.

Library for PIC

The library for PIC is provided only runtime library. The standard library function can not use.

<code>c900pic.lib</code>

PIC Caution

- PIC function can not be specified with inline function(`__inline`) to the same function.
- In the section which specified absolute address by `#pragma section code directive`, a specification of `__pic` qualifier become invalid.

3.14 PID Specifications

PID is the abbreviation for Position Independent Data, it is the data which can be accessed in same code without being dependent in any location on a memory.

Register for PID

When using PID variables, "XIY" register is used as the base register. Therefore, "XIY" register can not be used for any uses other than the base register.

A initial location address of PID is stored in "XIY" register.

Section Name

The section name in which PID variable is stored serves as the following sections according to the kind of variable. The PID variable can not be allocated into any sections other than these section names.

Table 3-13 Section Categorization of PID Variable

Section name	Items Subject to Categorization
pid_area	Data for which an initial specifier has not been specified
pid_data	Data for which an initial specifier has been specified
pic_data	Data for which rewrite is not performed

Moving PID

PID variables can be moved on memory while the program is executing, and it can also access PID variable at destination as it is. When moving PID variable, whole "pid_area" and "pid_data" sections must be moved, and the program which moves the "pid_area" and "pid_data" sections is needed.

PID Caution

- Between #pragma pid_on and #pragma pid_off, change to area, data and const sections by #pragma section directives cannot be performed.
- In the section which specified absolute address by #pragma section directives, a specification of corresponding __pid qualifier become invalid.

Part 4 Generating Execution Programs from the Command Line

Chapter 4 Compiler Driver Overview

The tools necessary for building an execution format program are all executed using a compiler driver (hereafter called a driver). As a result, execution of processes including compiling, assembling, and linking of a program from the command line is easy. With the driver, based on things like the specified option or input file suffix, activation of the compiler, assembler, etc. is controlled and output files are generated. When use the compiler or the assembler, rather than directly activating it, use each tool by using the driver.

Following are the four types of tools controlled by the driver.

- Compiler
- Macro Preprocessor
- Assembler
- Linker

The syntax for using the compiler driver is as follows.

cc900 [<options>] <files>

<options> shows the combination of options used by the driver. For detailed information on the options that can be specified with this compiler, see Chapter 5, "Option Descriptions".

<files> shows the files that are subject to compiling, assembling, and linking. The file extensions that are recognized by the driver as input files are as follows.

Table 4-1 Input File Types

Suffix	File Type
.c	C language source file
.i	Compiler preprocessor output file
.mac	Macro preprocessor source file
.asm	Assembler source file
.rel	Relocatable object file
.lib	Library file
.lcf	Link command file

Also, the files that are generated as a result of the aforementioned input files being processed by a compiler, assembler, or linker are as follows.

Table 4-2 Output File Types

Suffix	File Type
.i	Compiler preprocessor output file
.asm	Assembler source file
.rel	Relocatable object file
.med	Macro preprocessor list file
.lst	Assembler list file
.map	Map file
.abs	Absolute object file

Chapter 5 Option Descriptions

-# Display the tool that activates the link process from compiling

[Description Format]

-#

[Explanation]

- During the time from compiling to linking, the tools called by the compiler driver are displayed in the command line. Also, things such as the options transferred to each tool are displayed.
- With this option, the processes from compiling to linking are simply displayed, and actual compiling, assembling, and linking processes are not performed.

[Example]

Specify as noted hereafter.

cc900 -Nb1 -g file.c -#

-A Compile in accordance with ANSI specification

[Description Format]

-A

[Explanation]

- It compiles in accordance with ANSI standard.
If -A option does not specify, the integral promotion and the arithmetic conversion for multiplication and division which are defined in ANSI standard are not performed.

[Example]

Specify as noted hereafter.

cc900 -Nb1 -A file.c

-D Define the preprocessor macro

[Description Format]

-D <identifier>[=<substitution element>]

[Explanation]

- The preprocessing directive #define has the same effect as when written at the beginning of a source program.
- It is not possible to specify following cases.
 - When there is a blank character between <Identifier> and =
 - When there is a blank character between = and <Replacement Text>
- When only an <identifier> is specified, " 1 " will be specified in the <substitution element> part.

Table 5-1 The Meaning of Option in the Source File

Option specified	Meaning in the Source File
-D<identifier>=<substitution element>	#define <identifier> <substitution element>
-D<identifier>	#define <identifier> 1

[Example]

For example, for the option specification noted hereafter,

```
cc900 -Nb1 -DDEBUG -DBUFFER_MAX=256 file.c
```

when the following contents are described at the beginning of the source file, the same meaning results.

```
#define DEBUG 1
#define BUFFER_MAX 256
```

-E Output preprocessor processing results to standard output

[Description Format]

```
-E
```

[Explanation]

- Outputs the results of the C source file processed by the preprocessor to standard output.
- When multiple files are specified for input, the processing results of all files are output to standard output.
- When -E option is specified, only the preprocessor is activated. Because of this, compiling, assembling, and linking are not performed.
- -P option is a related option.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -E file1.c
```

-F Specify the value which fill a blank area

[Description Format]

```
-F<Value>
```

[Explanation]

- Fill a blank area in the output section with the specified value.
- Specify <Value> using a 2-byte constant.
- Fill a blank area in order of the upper byte and lower byte of the <Value>.
- When the size of the blank area is odd byte, the last 1 byte is initialized by the upper byte of the <Value>.
- It is only memory areas for which 'I' attribute is specified with the memory definition instruction of the link command file. Note that when the memory attribute specification is omitted, 'I' attribute is added by default.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -F0xffff file1.c
```

-I Specify Search Path for Include Files

[Description Format]

```
-I<Path>
```

[Explanation]

- Specifies the search path for the include files.
- In <Path>, it is possible to specify either an absolute path or a relative path, but this cannot be omitted.
- When multiple <Path> are specified with -I option, searching is done in the sequence in which they were specified.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -I..\user\original\file.c
```

-J Recognize the kanji code included in a source file

[Description Format]

```
-J
```

[Explanation]

- Recognizes the kanji code included in a source file.

[Additional Note]

Do not use this option for English version.

-L Specify a search path for input file to the linker

[Description Format]

```
-L<Path Name>
```

[Explanation]

- Specifies a search path for input files to the linker such as relocatable object files (.rel) or library files (.lib).
- In <Path Name>, it is possible to specify either an absolute path or a relative path, but this cannot be omitted.
- When a relative path is specified in <Path Name>, it is used as a relative path from linker executing directory.
- The file is searched in order of the following directories.
 - (1) The current directory.
 - (2) The directory specified with -L option (when specified more than once, the paths are searched in the order specified).
 - (3) The lib directory under the directory specified in the environment variable THOME900.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -L\usr\local\work file1.c file2.rel
cc900 -Nb1 -L..\work\ file1.c file2.rel
```

-Mi No Processing Macro Preprocessor**[Description Format]**

```
-Mi
```

[Explanation]

- Control the process so that Macro Preprocessor is not activated.

[Example]

In the following specification, file1.mac is processed in order of Assembler and Linker, but not by Macro Preprocessor.

```
cc900 -Nb1 -Mi file1.mac
```

-Nb Select CPU Type**[Description Format]**

```
-Nb[ <CPU Type> ]
```

[Explanation]

- Specifies CPU type of TLCS-900 family.
- <CPU Type> is omitted, it means that <CPU Type> is defined as 0.
- This option is omitted, it means that -Nb0 is specified.
- <CPU Type> are as follows.

Table 5-2 CPU Type

CPU Type	Function
0	TLCS-900 series
1	TLCS-900/L series, TLCS-900/L1 series
2	TLCS-900/H series
3	TLCS-900/H1 series

[Example]

In use of CPU of TLCS-900/L1 series, specify as noted hereafter.

```
cc900 -Nb1 file.c
```

-O Perform Optimization**[Description Format]**

```
-O[ <Optimization Level> ]
```

[Explanation]

- The level of the optimization which a compiler performs is specified for the numerical value of 0, 1, 2, or 3 to <Optimization Level>. The larger the number, the greater the optimization that the compiler performs, and the <Optimization Level> numerical value also includes optimization lower than that.
- When <Optimization Level> or this option is not specified, the compiler regards this as optimization level 0 having been specified.
- The optimization types performed with each optimization level are as follows. For detailed information on each optimization, see Chapter 6, "Optimizations".

Table 5-3 Optimization

Level	Function
0	Minimum optimization
1	Basic block optimization
2	Optimization of more than basic block
3	Maximum optimization

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -O3 file1.c
```

At this time, file.c is compiled at optimization level 3, and the optimization levels 1 and 2 optimizations are also done.

-P Execute only the preprocessor**[Description Format]**

```
-P
```

[Explanation]

- Outputs to a file the results of a source program processed by compiler preprocessing.
- Creates a file with the source file suffix changed to .i, and outputs the preprocessor execution results. The file name to which results are output can be changed using the -o option.
- When -P option is specified, only compiler preprocessing is performed. Therefore, compiling, assembling, and linking are not performed.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -P file1.c
```

-S Create assembly language source program**[Description Format]**

```
-S
```

[Explanation]

- Compiles a C source files or processes a macro preprocessor source file with macro preprocessor, and creates an assembly language source program.

- Because only compile processing or macro preprocess processing is performed, assembling and linking are not performed.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -S file1.c
```

-U Disable the specified preprocessor macro definition

[Description Format]

```
-U<identifier>
```

[Explanation]

- The preprocessing directive `#undef` has the same effect as when written at the beginning of a source program.
- When specified at the same time as `-D`, the option described later is enabled.
- This option does not have an effect on macros defined by the preprocessing directive `#define` in the source program.

Table 5-4 The Meaning of Option in the Source Program

Option specified	Meaning in the Source Program
<code>-U<identifier></code>	<code>#undef<identifier></code>

[Example]

Specify as noted hereafter,

```
cc900 -Nb1 -UDEBUG file1.c file2.c
```

the option specification noted above has the same effect as when

```
#undef DEBUG
```

is described at the beginning of `file1.c` and `file2.c`.

-V Output version information

[Description Format]

```
-V
```

[Explanation]

- Outputs version information to standard output.
- This option cannot be included in files for which `-f` option is specified.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -V
```

-W Transfer option to specified tool**[Description Format]**

```
-W<Sub-option>,<Option>[,<Option>...]
```

[Explanation]

- This option transfers specified options to specific tools such as compiler, assembler, linker, or Macro Preprocessor specified using <Sub-option>. The types of <Sub-options> are as follows.

Table5-5 Sub-options

<Sub-option>	Tool
0	Parser (for C compiler)
2	Code Generator (for C compiler)
a	Assembler
l	Linker
m	Macro Preprocessor

Normally, for options specified to the compiler driver, the options are allocated to tools such as the compiler, assembler, linker, or Macro Preprocessor by the judgment of the driver, so this option is not used.

[Example]

When specified as follows, the driver transfers -O0 option to assembler. This has deterred optimization of assembler.

```
cc900 -Nb1 -Wa,-O0 file.c
```

-XA, -XC, -XD Change the Section Alignment**[Description Format]**

```
-XA<Alignment>
-XC<Alignment>
-XD<Alignment>
```

[Explanation]

- Changes the alignment of area, const, or data section. This option is effective to start location of section and allocation alignment.

Table5-6 Subject of Change for Each Option

Option	Subject of Change	Default Alignment
-XA	area section	2 byte
-XC	const section	2 byte
-XD	data section	2 byte

- <Alignment> specifies 1 or 2.

[Example]

Specify as noted hereafter, when the alignment of area section and data section become 1 byte.

```
cc900 -Nb1 -XA1 -XD1 file.c
```

-XE Handles \ in assembler source files as a character**[Description Format]**

-XE

[Explanation]

- Handles the escape character " \ " included in assembler source files as a character rather than as an escape sequence start symbol.
- When the input file is an assembler source file, the driver activates assembler. When escape characters are included in this assembler source file, specify this option.
- When performing compiling, assembling, or linking from a C language source file, the driver automatically transfers this option to assembler, so the user does not need to pay special attention to this issue.

[Example]

For example, when the escape character " \ " is included in assembler source files, specify as follows.

cc900 -Nb1 -XE file.asm

Note that unless an escape sequence is described in an invalid position, compiler is able to process correctly, so it is not necessary to specify this option.

-XF Leave an assembly language source program**[Description Format]**

-XF

[Explanation]

- Leaves an assembly language source program created at the midway state of compiling, assembling, and linking without deleting.
- When this option is not specified, an assembly language source program in the middle of compile processing is deleted.
- -S option is a similar option. -S option does not execute assembling and linking processing.

[Example]

Specify as noted hereafter.

cc900 -Nb1 -XF file.c

-XS Output the code by size optimization**[Description Format]**

-XS

[Explanation]

- Output the code that object size becomes small. Although the memory utilization improves, execution speed may deteriorate.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -XS file.c
```

-Xaa, -Xac, -Xad Create the section for 1 byte variables**[Description Format]**

```
-Xaa
-Xac
-Xad
```

[Explanation]

- Create area, const, or data section for 1 byte variables. The 1 byte variable is packed the section of the following section name.

Table5-7 Subject of Change for Each Option

Option	Target Section	Default Displacement	Section Name
-Xaa	area section	far	f_area_align1
		near	n_area_align1
		tiny	t_area_align1
-Xac	const section	far	f_const_align1
		near	n_const_align1
-Xad	data section	far	f_data_align1
		near	n_data_align1
		tiny	t_data_align1

- The alignment of the section for 1 byte variable is 1 byte. The section The start location alignment of section and each data alignment in section is specified 1 byte.
- Change the default displacement by -ZA, -ZC or -ZD option.

[Example]

Specify as noted hereafter, 1 byte variable as which the initializer is not specified is packed as f_area_align1 section.

```
cc900 -Nb1 -Xaa file.c
```

-Xc Specify the function type to cdecl**[Description Format]**

```
-Xc
```

[Explanation]

- Specify the function type to cdecl.
- This option has same effect as specifying __cdecl to each function of a file or writting "#pragma cdecl" at the beginning of a source file.

- If -Xc or -Xr options is not specified, the default function type is cdecl.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -Xc file.c
```

-Xec Change the type of the enumeration constant

[Description Format]

```
-Xec
```

[Explanation]

- Determines the type of enumeration constant to the scope of enumerator values. The type of enumeration constant is selected in following order.

Table 5-8 Enumeration Constant Type Sequence

Value Range	Type of Enumeration Constant
0 - 255	unsigned char
-128 - +127	signed char
0 - 32767	unsigned int
-32768 - +32767	signed int

- If the value is out of range listed above, an error is outputted.
- This option cannot be specified with -A option at the same time.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -Xec file.c
```

-Xns Deter Integration of Stack Freeing

[Description Format]

```
-Xns
```

[Explanation]

- Deter the optimization which stack freeing collectively. Specifying this option helps to increase RAM efficiency. However code efficiency and execution speed may deteriorate.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -Xns file.c
```

-Xp Packing the structure

[Description Format]

`-Xp<Alignment>`

[Explanation]

- Packing the structure, and change the alignment of structure member.
- `<Alignment>` specifies 1 or 2.

[Example]

Specify as noted hereafter, the alignment of structure member become 1 byte.

`cc900 -Nb1 -Xr file.c`

-Xr Specify the function type to adecl

[Description Format]

`-Xr`

[Explanation]

- Specify the function type to adecl.
- This option has same effect as specifying `__adecl` to each function of a file or writting "#pragma adecl" at the beginning of a source file.
- If `-Xc` or `-Xr` options is not specified, the default function type is `cdecl`.
- This option invalid to the standard library.

[Example]

Specify as noted hereafter.

`cc900 -Nb1 -Xr file.c`

-Xsc Change the processing method of shift operation

[Description Format]

`-Xsc`

[Explanation]

- When the number of bits to shift is larger than the bit width of the variable shifted, the operation result of an arithmetic shift to the left is set to 0, and the case of the operation result of an arithmetic shift to the right becomes the value fill used with the sign bit about all the bits.

[Example]

Specify as noted hereafter.

`cc900 -Nb1 -Xsc file.c`

-Xub Handle bit field members as unsigned**[Description Format]**

```
-Xub
```

[Explanation]

- Handles the bit field member which is declared without adding a type specifier "signed" or "unsigned" as "unsigned".
- If this option is not specified, the bit field member which is declared without adding a type specifier "signed" or "unsigned" are handled as "signed".

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -Xub file.c
```

-Xuc Handle the char type as unsigned char type**[Description Format]**

```
-Xuc
```

[Explanation]

- Handles the char type which declared without adding a type specifier "signed" or "unsigned" as unsigned char type.
- If this option is not specified, the char type which declared without adding a type specifier "signed" or "unsigned" are handled as signed char type.

[Example]

For example, when compiling is done with -Xuc option added to the source file called

```
char message[] = "Hello World!"
```

then this is handled as:

```
unsigned char message[] = "Hello World!"
```

-Xw Change the bit field allocation sequence**[Description Format]**

```
-Xw
```

[Explanation]

- Changes the bit field memory allocation sequence from the least significant bit(LSB).

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -Xw file1.c
```

-ZA, -ZC, -ZD, -ZT Change the default displacement**[Description Format]**

```
-ZA <tiny|near|far>
-ZC <near|far>
-ZD <tiny|near|far >
-ZT <near|far>
```

[Explanation]

- Changes the displacement of an area, const, data, or code section.

Table 5-9 Subject of Change for Each Option

Option	Subject of Change
-ZA	area section
-ZC	const section
-ZD	data section
-ZT	code section

- Specifying displacement using #pragma command or function qualifier has higher priority than using option, so when there is specification of displacement using #pragma directive or function qualifier in source program, the specification using this option is disabled for parts that are subject to these.

[Example]

To make the displacement for area sections and data section to "tiny", specify as follows:

```
cc900 -Nb1 -ZAtiny -ZDtiny file.c
```

-Za, -Zc, -Zd, -Zt, -Zi Change the section name and the displacement**[Description Format]**

```
-Za <section name> [,<tiny|near|far>]
-Zc <section name> [,<near|far>]
-Zd <section name> [,<tiny|near|far >]
-Zt <section name> [,<near|far>]
-Zi <section name>
```

[Explanation]

- Changes the section name and the default displacement of area, const, data, or code section. The io section can change the section name only.

Table 5-10 Subject of Change for Each Option

Option	Subject of Change
-Za	area section
-Zc	const section
-Zd	data section
-Zt	code section
-Zi	data section

- You cannot use an assembler reserved word for <section name>. The <section name> must be a name that complies with the identifier description rules for the assembly language.
- When <displacement> is omitted, only the section name is changed.

[Example]

When specified as follows, the name of area section included in file1.c is changed to "New_name", and the section displacement is changed to "tiny".

```
cc900 -Nb1 -Za New_name,tiny file.c
```

-c Create a relocatable object**[Description Format]**

```
-c
```

[Explanation]

- Executes compiling and assembling and creates a relocatable object file (suffix .rel).
- Linking is not executed.
- When multiple source programs are specified, compiling and assembling for each are executed, and a relocatable object file corresponding to each source program is created.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -c file1.c file2.c
```

-e Create an error list file**[Description Format]**

```
-e<File Name>
```

[Explanation]

- The errors and the warnings found by each tool such as the compiler, assembler, and linker are consolidated and output to a specified file.
- When Fatal Error occurs, an error list file cannot be created.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -e errlst.txt file1.c
```

-f Read the file that describes the option to be used**[Description Format]**

```
-f <File Name>
```

[Explanation]

- Describes in advance in a text file the option to be specified. By using the -f option to specify that file, options are read automatically.
- It is possible to describe options on multiple lines within a file which is specified as <file name>.
- -V and -f options cannot be written in a file which is specified as <file name>.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -f optionlist.txt file1.c file2.c
```

-g Outputs debugging information

[Description Format]

```
-g
```

[Explanation]

- Outputs debugging information to an object file.
- It is possible to perform debugging by reading a created object to debugger.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -g file1.c file2.c
```

-l Create a list file

[Description Format]

```
-l <Sub-option>
```

[Explanation]

- Outputs each type of information to a list file. The type of information output to the list file is specified using a sub-option. Multiple sub-options can be specified.
- List files are output with different extensions attached for each tool, and the contents also differ for each list file. About the contents of the list file, refers the "TLCS-900 Assembler Reference".
- The relationship between each tool and output suffix is as noted below.

Table 5-11 Tools and Extensions

Tool	Suffix
Assembler	.lst
Linker	.map
Macro Preprocessor	.med

- Multiple sub-options can be specified.

Table 5-12 Sub-options and Contents

Sub-option	Contents
None	Outputs basic information to a list file.
a	Outputs the link information of external variables and local variables to a list file. (for Linker)
f<File Name>	Outputs by adding a name to the list file.
s	Outputs the link information of static identifier to a map file (for Linker). It is necessary to specify -g option at compiling.
x	Outputs cross reference information to a list file.

[Basic Information]

- Activated tool's path, environmental variable, linked file list, created list files, option list that contains all created objects, and link command file contents

[Example]

Specify as noted hereafter, when the link information of the external variables and local variables output.

```
cc900 -Nb1 -la file.c
```

Specify as noted hereafter, when all link information output.

```
cc900 -Nb1 -g -la -ls file2.c
```

-o Specify an output file name**[Description Format]**

```
-o <File Name>
```

[Explanation]

- Specifies a output file name using <File Name>.
- When this option is not specified, the input file name with a changed suffix becomes the output file name.

[Example]

For example, when specified as follows,

```
cc900 -Nb1 -o output.abs input.c
```

the name of the absolute object file is output as output.abs.

-r Perform incremental linking**[Description Format]**

```
-r
```

[Explanation]

- For link processing, performs incremental linking.
- The output file is a relocatable object file.
- When this option is specified, specify -o option at the same time and be sure to specify an output file name.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -r -o outfile.rel file1.c file2.rel
```

-s Perform Macro Preprocessor Identifier definition

[Description Format]

`-s <Identifier>[=<Value>]`

[Explanation]

- Defines <Value> to <Identifier> used by Macro Preprocessor.
- This option has the same effect as when "variable" function of Macro Preprocessor is used.
- If <Value> is omitted, it means that <Value> is defined as 1.
- It is not possible to specify following cases.
 - When the <Value> is not an integer
 - When there is a blank character between <Identifier> and =
 - When there is a blank character between = and <Value>

[Example]

Specify as noted hereafter.

`cc900 -Nb1 -sAAA=1 file.mac`

-u Ignore predefined macros

[Description Format]

`-u`

[Explanation]

- Ignores predefined macros the following.
 `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`

[Example]

Specify as noted hereafter.

`cc900 -Nb1 -u file.c`

-u Register undefined symbols to the symbol table

[Description Format]

`-u<Symbol>`

[Explanation]

- Linker links specified <Symbol> forcibly.
- This option is used in order to force a modular link. For example, this is used to link a library function that is not being used.
- This option can specify multiple.
- Do not put blank character between -u and <Symbol>.

[Example]

When specified as follows,


```
cc900 -Nb1 -u_pow file.c
```

even if `pow()` function is not used within `file.c`, linker links `pow()` function from a standard library.

-w Set the Warning Level**[Description Format]**

```
-w<Warning Level>
```

[Explanation]

- Specifies the warning level which output by compiler, assembler, or linker.
- <Warning Level> can be specified from 0 to 3, and warning is output so more strictly that the number becomes large.
- When this option is omitted, regards as `-w1` being specified.
- When <Warning Level> is omitted, regarded as `-w0` being specified. No warning is outputted at this time.

[Example]

Specify as noted hereafter.

```
cc900 -Nb1 -w3 file.c
```


Part 5 Optimizations

Chapter 6 Optimizations

6.1 Optimizations by C compiler

TLCS-900 family C compiler performs optimizations in order to improve object efficiency. The optimization level of compiler is controllable by specifying -O option. The optimization level detail is as follows.

Level 0	Minimum optimization
Level 1	Weak optimization
Level 2	Strong optimization
Level 3	Maximum optimization

For example, if -O2 option is specified, C compiler performs optimization to level 2. If -O3 option is specified, all optimization are performed. C compiler applies the methods of all the optimization explained by the following clause.

In addition, execution speed may improve by changing the description of a program other than the optimization which the C compiler performs. For details, see Section 6.2, "Optimizations by User".

6.1.1 Assignment of Variables to Registers

Auto variables are assigned to registers regardless of the optimization level. Variables are assigned to registers as much as possible to improve code efficiency and execution speed.

With the TLCS-900 family C compiler, even if a storage class for register is not specified in a source program, the variable are assigned to a register if it is possible to assign.

When there are variables which are not used simultaneously, it may be assigned to a same register. It makes efficient use of registers. For example, local variables i and j are assigned to registers.

```
void makeunit(int *mat)
{
    int i, j;

    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            if (i == j) {
                *(mat + 4 * i + j) = 1;
            }
            else {
                *(mat + 4 * i + j) = 0;
            }
        }
    }
}
```

Since optimization of assignment to registers corresponds to source level debugging, the user can debug at source level without worrying about this optimization.

6.1.2 Integrated Stack Release

This processing is performed regardless of the level of optimization. When cdecl functions, which pass arguments via the stack, are used consecutively (see __cdecl in 3.2.1 "

Function Qualifier"), the arguments are released from the stack all at once.

Usually, functions are called using the following procedure:

1. Arguments and return address are pushed on to the stack.
2. The function is called.
3. The pushed arguments are popped from the stack (stack release).

The stack releasing, the number of bytes accumulated on the stack as argument is added to the stack pointer.

When function call is continuing, stack release all at once improve code efficiency and execution speed, compared with stack release for every function call. When you deter this optimization, specify -Xns option.

6.1.3 Minimum Optimization

The following optimizations are performed.

- Meaningless instructions elimination
- Change to high-functional or shorter instructions.

6.1.4 Branch Optimizations

Unnecessary branch instructions are deleted by branch optimizations. As a result of this optimizing, there are lines which are not executed in the source program. Note that a break point cannot be specified in lines which are not executed at source-level debugging. The technique of branch optimizations is as following:

Switch statements

When there are six or more case statements, and the interval of labels is continuous or narrow, a jump table is generated. This reduces conditional branch and improves speed.

Deletion of redundant branch instructions

Branch instructions used to jump to the next instruction are deleted. Such instructions may be generated due to other optimization (such as the deletion of unnecessary instructions). Redundant labels resulting from optimization are also deleted.

Deletion of codes which are not executed

Instructions which are not controlled from the program flow are deleted. For example, codes following an unconditional branch are not executed, so they are deleted. Redundant branch and labels generated as a result are also deleted.

Deletion of instruction to branch to function exit

When the branch destination by a branch instruction was made into the exit of a function, the function exit processing is performed in the location. Thereby, it may become processing more nearly high speed than branch. This optimization is not performed if -XS option is specified (code size has a high priority at code generation).

6.1.5 Unnecessary Instructions Elimination

Unnecessary instructions are deleted by this optimization. That is, definition instruction which set value to unused variables are deleted.

In the example below, if auto variable i is not used anywhere else, line "i = 1; " is unnecessary, so it is not coded. A break point cannot be set on this line by the source-level debugger.

```

void func(void)
{
    int    i;
    ...
    i = 1; /* This line is unnecessary, so it is deleted. */
    return;
}

```

6.1.6 Copy Propagation

Copy propagation is performed by optimization level over the narrow range or the whole function. Copy propagation means the processing which reuse variables that become same value by copying or assignment between variables and reduces the variable in subsequent expression.

Variables are reduced as follows:

- Propagation of value obtained by copying variable
- Propagation of assigned constant value

Variables propagation:	
x = y;	>>> x = y;
u = x * y;	u = x * x;
Constant values propagation:	
x = 1;	>>> x = 1;
y = 2;	y = 2;
z = x + y;	z = 3;
x += y;	x = 3;

If the condition for a conditional branch is constant-constant comparison as a result of copy propagation, the conditional branch is either deleted or changed to an unconditional branch.

Copy propagation may change variables used in the source program to constants or replace them. Therefore, variables in the source program cannot be used at source-level debugging, or the order in which an expression is evaluated may change. Note this at source-level debugging. In order to deter the optimization to a variable, specify a volatile qualifier to a variable.

6.1.7 Common Sub-expression Elimination

This process is performed by optimization level over the narrow range or the whole function.

Common sub-expression elimination means that if there is sub-expression which is performed same calculation in multiple expressions, the sub-expression is evaluated once to reduce the number of operations.

```

x = 4 * i + j;      >>>  t0 = 4 * i;
y = 4 * i - j;      x = t0 + j;
                    y = t0 - j;
/* t0 is a variable generated by compiler, usually register is used.
*/

```

The number of times of operation becomes fewer, so that there are many common sub-expressions.

However, the order in which an expression is evaluated may change, note this at source-level debugging.

6.1.8 Loop Optimizations

The loop optimization is performed. The C compiler uses two methods for loop optimization as following.

Move loop invariant

It is the processing which moves the invariant in a loop out of a loop, and reduces execution time of loop.

```
for (i = 0; i < 10; i++)    >>>    t0 = b + c ;
    a[i] = b + c;                for (i = 0; i < 10; i++)
                                a[i] = t0;
/* t0 is a variable generated by compiler, usually register is used. */
```

Optimization of induced variables

It is the processing which finds the changing variables a steady value in the loop, and simplifies calculation. This method is used in address calculation for an array, for example.

```
for (i = 0; i < 10; i++)    >>>    t0 = &a[0];
    a[i] = 0;                for (i = 0; i < 10; i++) {
    .                        *t0 = 0;
    .                        ++t0;
                                }
/* t0 is a variable generated by compiler, usually register is used. */
```

The variables in the source program cannot be used at source-level debugging, or the order in which an expression is evaluated may change by these optimizations. Note this at source-level debugging.

6.2 Optimizations by User

In this section, we will explain optimization methods when describing source program and an option used to improve the efficiency of object code without using -O option.

6.2.1 Optimizations when Describing Source Program

When describing source program, note the followin.

Passing arguments via register

When passing arguments to a function, using registers instead of the stack reduces the stack usage and execution speed may improve. This is performed by setting the function type to `adecl`, that is specifying -Xr option, or `#pragma adecl` directives, or function qualifier `__adecl`. (See 3.2.1 "Function Qualifier")

Using inline functions

Using inline functions reduces overhead caused by a function call and generates object code which operates fast. At the same time, the code size may become large. To determine which functions should be inline, check the software environment. This is performed by specifying the function qualifier `__inline`.

6.2.2 Optimizations by Option

There are the following options which improve object efficiency besides -O option.

Specification of Size Priority Code Generation

In generating code, This C compiler's priority is usually speed. Specifying -XS option, changes the code-generation priority to size. However, optimization is easier for code where the priority is speed.

Thus, the code size where the priority is speed may be smaller than for code where priority is size. To reduce the size of the object, try to use different combinations with `-XS` and `-O` options.

Part 6 Standard Library Functions

Chapter 7 Standard Library Overview

TLCS-900 family C compiler shall generate objects that performed in the free standing environment. The free standing environment means the environment performs the C programs operation system support. This is defined in the Programming Language C(ISO/IEC 9899 : 1990) standard. In the free standing environment, provided library functions are implementation-defined.

In this chapter, it explains the header files and the library functions provided by this compiler.

Header Files

The header files which this compiler offers are stored in the include directory in the install directory.

Library Files

The library functions which this compiler offers are stored as the library files in the lib directory in the install directory.

Chapter 8 Header File

The header files of the standard library which this compiler offers are as follows.

Table 8-1 Standard Library Header File

Header File Name	Header File Contents
assert.h	assert
ctype.h	Character type
errno.h	Error processing
float.h	Floating point type quantitative limit
limits.h	General integer type quantitative limit
math.h	Standard numerical value mathematical function
setjmp.h	Global jump
stdarg.h	Variable count argument list
stddef.h	Library type and macro
stdio.h	Input/output
stdlib.h	Utility function
string.h	Character string operation

This library is not support <setjmp.h>.

Chapter 9 Library Function

9.1 Standard Library Function List

The standard library functions which this compiler offers are as follows.

<assert.h>

function name	function
assert()	macro for program assert

<ctype.h>

function name	function
isalnum()	judge the character as alphabet or number
isalpha()	judge the character as alphabet
isctrl()	judge the character as control character
isdigit()	judge the character as decimal number
isgraph()	judge the character as display characters except for blanks
islower()	judge the character as lower case alphabet
isprint()	judge the character as display characters
ispunct()	judge the character as delimiter
isspace()	judge the character as blank type characters
isupper()	judge the character as upper case alphabet
isxdigit()	judge the character as hexadecimal number
tolower()	convert upper case alphabet to lower case
toupper()	convert lower case alphabet to upper case

<Notice>

The standard library functions of <ctype.h> are offered even when it is macro besides a function.

When you use functions, define the macro name "__CTYPE_FNC".

```
#define __CTYPE_FNC
#include <ctype.h>
```

When you use macros, a character string is restricted to 256 bytes.

Although a function or a macro can be specified for every file, in order to avoid confusion, we recommend you to use either.

<math.h>

function name	function
acos()	calculate the inverse arc cosine
asin()	calculate the inverse arc sine
atan()	calculate the inverse arc tangent
atan2()	calculate the inverse arc tangent2
ceil()	return the minimum integer that is not smaller than the argument
cos()	calculate the cosine
cosh()	calculate the hyperbolic cosine
exp()	calculate the exponential
fabs()	calculate the floating-point absolute value
floor()	return the largest integer value which does not exceed the value of argument
fmod()	calculate floating-point remainder
frexp()	separate floating-point value into mantissa and exponent
ldexp()	calculate a real number from mantissa and exponent
log()	calculate natural logarithmic
log10()	calculate common logarithmic
modf()	separate floating-point value into integer part and fractional portion
pow()	calculate the power
sin()	calculate the sine
sinh()	calculate the hyperbolic sine
sqrt()	calculate the square root
tan()	calculate the tangent
tanh()	calculate the hyperbolic tangent

<stdarg.h>

function name	function
va_arg()	Return a value of actual argument from a variable arguments list
va_end()	End a handling of a variable argument list
va_start()	Initialize a variable argument list

<Notice>

In the function with a variable argument, a float type is extended to a double type and a stack is loaded with it. Therefore, a float type argument cannot be referred to correctly with va_arg() function. Please do not use a float type for a variable argument.

<stdio.h>

function name	function
sprintf()	Write formatted data to a character string
sscanf()	Read formatted data from a character string
vsprintf()	Output argument list value to a character string buffer

<stdlib.h>

function name	function
calloc()	allocate memory dynamically and clear it
malloc()	allocate memory dynamically
realloc()	reallocate memory dynamically
free()	deallocate memory
atof()	convert character string to numerical value of double type
atoi()	convert character string to integer value of int type
atol()	convert character string to integer value of long type
strtod()	convert character string to numerical value of double type
strtol()	convert character string to integer value of signed long type
strtoul()	convert character string to integer value of unsigned long type
bsearch()	binary search in an array
qsort()	sort an array
abs()	calculate integer absolute value
div()	calculate quotient and remainder
labs()	calculate absolute value of long type integer
ldiv()	calculate quotient and remainder by performing division long type integer
srand()	initialize the pseudo-random number
rand()	generate the pseudo-random number

<string.h>

function name	function
memchr()	search character from memory area
memcmp()	compare memory block
memcpy()	copy memory block
memmove()	move memory block
memset()	set memory to specific value
strcat()	concatenate 2 character strings
strchr()	search location of character of character string
strcmp()	compare 2 character strings
strcpy()	copy character string
strcspn()	search a portion which does not contain character set from character string
strlen()	return character string length
strncat()	concatenate n character
strncmp()	compare n character
strncpy()	copy n character
strpbrk()	search character of charset from character string
strrchr()	search location of character from character string
strspn()	search a character set from character string
strstr()	search location of part of character string
strtok()	extract a next token from character string

<Notice>

To use a floating point number, it is necessary to define a global variable "errno".

9.2 Runtime Library List

The object which this compiler outputs calls the following runtime libraries.

Table 9-1 Runtime Library List

Runtime Library Name	Function
C9H_mulls	multiplication operation of signed long
C9H_divls	division operation of signed long
C9H_remlse	remainder operation of signed long
C9H_mullu	multiplication operation of unsigned long
C9H_divlu	division operation of unsigned long
C9H_remlu	remainder operation of unsigned long
_fneg_f	negation operation of float
_fneg_d	negation operation of double
_fneg_x	negation operation of long double
_fadd_f	addition operation of float
_fadd_d	addition operation of double
_fadd_x	addition operation of long double
_fsub_f	subtraction operation of float
_fsub_d	subtraction operation of double
_fsub_x	subtraction operation of long double
_fmul_f	multiplication operation of float
_fmul_d	multiplication operation of double
_fmul_x	multiplication operation of long double
_fdiv_f	division operation of float
_fdiv_d	division operation of double
_fdiv_x	division operation of long double
_fld_ff	assignment operation of float
_fld_fd	conversion and assignment operation from double to float
_fld_fx	conversion and assignment operation from long double to float
_fld_df	conversion and assignment operation from float to double
_fld_dd	assignment operation of double
_fld_dx	conversion and assignment operation from long double to double
_fld_xf	conversion and assignment operation from float to long double
_fld_xd	conversion and assignment operation from double to long double
_fld_xx	assignment operation of long double
_fld_lf	conversion and assignment operation from float to long
_fld_ld	conversion and assignment operation from double to long
_fld_lx	conversion and assignment operation from long double to long
_fld_ulf	conversion and assignment operation from float to unsigned long
_fld_uld	conversion and assignment operation from double to unsigned long
_fld_ulx	conversion and assignment operation from long double to unsigned long
_fld_fl	conversion and assignment operation from long to float
_fld_dl	conversion and assignment operation from long to double
_fld_xl	conversion and assignment operation from long to long double
_fld_ful	conversion and assignment operation from unsigned long to float
_fld_dul	conversion and assignment operation from unsigned long to double
_fld_xul	conversion and assignment operation from unsigned long to long double
_fcmp_f	comparison operation of float
_fcmp_d	comparison operation of double
_fcmp_x	comparison operation of long double
_fcmp_zf	comparison operation of float and int
_fcmp_zd	comparison operation of double and int
_fcmp_zx	comparison operation of long double and int

<Notice>

Assembly language program should not define the identifier of the same name as these libraries.

Part 7 Error Message

Chapter 10 Error Message Format

10.1 Types of Error Messages

There are the following three types of error message.

Warning

A warning is output when a compile result may become what a user does not mean. The compiler outputs a warning, but compiling work continues, and output file is generated.

Error

An error is output when syntax that violates the rules is detected.

Fatal Error

A fatal error is output when some kind of serious problem occurs under compiling and it is no longer possible to progress with source file compiling.

10.2 Error Message Format

Error messages take the following format:

<Filename> <Line Number> : <Tool>-<Type>-<Number> : <Message>

<Filename>	This is the name of the file in which the error occurred. <filename> is not output when the cause of an error is not related to a specific file.
<Line Number>	This is the number of the line in which the error occurred. Usually, it outputs to an error as which a filename is output.
<Tool>	This is the name of the tool in which the error is occurred.
<Type>	This is the error type. Fatal Error Warning
<Number>	This is the error number, described later. Fatal : 0 - 99 Error : 200 - 499 Warning : 500 - 999
<Message>	The message is a description of the error.

Chapter 11 Driver Error Message

11.1 Driver Fatal Error

<I/O Errors>

- 20:** *Can't open "<filename>"*
 The driver cannot open the specified file.

<Invocation Errors>

- 100:** *No source file found in invocation*
 No input source file was specified in the command.
- 103:** *"<filename>" files are the same*
 The same filename is specified more than once.
- 106:** *Missing parameter "<option>"*
 A parameter which should be specified for an option is missing.
- 109:** *Unrecognized option "<option>"*
 An invalid option is specified.
- 111:** *Can't nest a command file*
 The option list file is nested.
- 112:** *Not allowed character "<option>"*
 The option parameter is not a numerical value.
- 113:** *Invalid subargument "<option>"*
 The method of specifying arguments necessary for options is not perfect.
- 114:** *Invalid argument "<option>"*
 There is an error in the method of specifying arguments necessary for options.
- 115:** *'-r' option requires '-o' option*
 You must specify a file output with the '-o' option when specifying option '-r'.
- 116:** *Can't execute "<filename>"*
 The program indicated with <filename> cannot be executed. Check that the execution procedure is correct and that the correct installation directory is set in the environment variable.

11.2 Driver Warning

- 520:** *The suffix not fit for output-file "<filename>"*
 There is an error in specified output file suffix.
- 521:** *Ignored option "<option>"*
 The opposite option was specified. Priority is given to the option specified first.
- 522:** *Unknown suffix, "<filename>" used as "<extension>" file*
 Error in specified input file suffix "<extension>".

Chapter 12 C Compiler Error Message

12.1 C Compiler Fatal Error

<Runtime errors>

100: *Too many errors*

The number of errors exceeded the compiler limit (30).

<File-related errors>

120: *Cannot close file "<filename>"*

The specified file cannot be closed.

121: *Cannot open file "<filename>"*

The specified file cannot be opened.

122: *Cannot open temporary file*

An intermediate file during compiling cannot be generated.

123: *Cannot seek "<filename>"*

The specified file cannot be sought.

125: *Problems with output file, probably out of disk space*

An error occurred during file write. Available space in disk may not be enough.

<Compiler limits>

130: *Compiler limit, out of space*

The memory which the compiler allocated reached the limit. The compiler cannot be allocated memory any more.

131: *Compiler limit, too deep nesting of blocks*

The number of nested blocks in the source program exceeded the compiler limit.

134: *Compiler limit, too deep nesting of struct/union*

The number of structure/union nesting levels exceeded the compiler limit.

136: *Compiler limit, too many internal variables in 'function'*

The number variables or types exceeded the compiler limit.

137: *Compiler limit, too many internal label*

The number of internal variables exceeded the compiler limit because the function is too large.

138: *Yacc stack overflow*

Compiling the source program cannot continue because the work area for analyzing the source program is insufficient.

<Fatal errors due to memory insufficiency>

140: *Out of memory*

The area necessary for compiling cannot be obtained because system memory is not sufficient.

142: *Too large function for optimization in '<function_name>'*

Memory for optimization cannot be obtained.

<Limits exceeded>

150: *Too large string*

The length of a character string exceeded the size of the compiler buffer.

151: *Too large string for inline assemble*

The character string specified in the inline assembly statement is too large.

<Other fatal errors>**160: *Division by zero in '<function>'*****161: *Remainder by zero in '<function>'***

Division by zero or remainder by zero is performed in the source program.

162: *Memory control blocks destroyed*

OS memory control blocks are destroyed.

163: *Unexpected EOF in comment*

The file ended before the end of a comment.

12.2 C Compiler Error

<Token errors>**200: *Empty character constant***

Null character constant was used as a character constant. A character constant must contain one character which is enclosed by single quotation.

201: *Illegal character '<Hexadecimal>' at column '<column>'*

A character whose character code is hexadecimal appeared in the source file. The character cannot be used.

202: *Illegal digit '<character>' for base '<radix>'*

A character whose character code is a Cardinal number(8 or 10 or 16) appeared in the source file. The character cannot be used.

203: *Illegal escape sequence*

An escape sequence is used in other than a character constant or a string literal.

204: *Illegal hex constants*

No hexadecimal character is written after 0x or \x indicating the start of a hexadecimal constant.

205: *Newline in char constant*

A newline character is input before the end of a character constant (before closing using " ' "). Lines cannot be changed within a character constant.

206: *Newline in string*

A newline character is input within a string literal.

207: *Unexpected exponent character '<character>'*

A character other than a sign or numerical value is written after character "e" or "E" representing exponent character.

<Syntax errors>**210: *Constant expected***

A constant expression is required. For example, an expression other than a constant expression is written instead of an array size for array declaration.

```
int a = 5;
int b[a];      /* incorrect */
```


211: *Illegal break*

A break statement is written in a statement other than a do, for, while, or switch iteration statement.

212: *Illegal continue*

A continue statement is written in a statement other than a do, for, or while iteration statement.

213: *Initialization needs {} in '<identifier>'*

An initializer for an array, structure, or union must be enclosed by braces {}.

```
int a[1] = 1;           /* incorrect */
int b[1] = { 1 };      /* correct */
```

215: *Syntax error in '<token>'*

A syntax error occurred at '<token>'.

216: *Syntax error at or near column <column>*

A syntax error occurred at or near the number of columns <column>.

<Declaration and definition errors>**221: *Array of functions not allowed***

An array of functions is not allowed. Nor can an array of void type be accepted. The pointer type to the function may have been incorrectly declared. The following shows an example of a pointer array to the function that returns the int type.

```
int (*f1[10])();
/* Correct: Pointer array to the function that returns
   the int type */
int *f2[10]();
/* Incorrect: Array of functions that return a pointer
   to int */
```

223: *Cannot initialize extern '<token>' in block-scoped*

A variable declared together with storage-class specifier extern within a block is initialized.

224: *Cannot use address of automatic variable as static initializer*

An address of an object with automatic storage duration is used within an initializer to an object with static storage duration.

```
void func()
{
    int i;
    int *p1 = &i;           /* correct */
    static int *p2 = &i;     /* incorrect */
```

225: *Duplicate signed/unsigned keywords*

Both "signed" and "unsigned" are used within one declaration.

226: *Duplicate storage class '<class>' specified*

Two or more storage-classes are specified within one declaration.

227: *Expected formal-parameter list*

An argument list instead of a parameter list is used for function definition.

```
/* correct */           /* incorrect */
void func(int arg)      void func(int)
{                       {
```

228: Function illegal in struct/union '<identifier>'

A function is declared as a structure/union member.

229: Illegal bit field type '<bit filed>'

Illegal bit field type (such as pointer or floating point type) is specified.

230: Illegal declaration '<storage class specifier>'

The specified storage-class specifier cannot be used.

231: Illegal function return type, cannot return array type

Array type is specified as function return value type. Pointer type to an array can be specified as return value type.

232: Illegal function return type, cannot return function type

Function type is specified as function return value type. Pointer type to a function can be specified as return value type.

233: Illegal initialization

Initialization specification is illegal.

235: Illegal type combination '<type specifier>'

Some type specifiers cannot be used within the same declaration.

236: Illegal void type '<identifier>'

An attempt was made to declare a void type variable. Declaration of pointer (incomplete type) to a void type variable or void can only be used to declare a function without a return value or to indicate no arguments for function declaration.

237: Illegal zero sized member '<member>'

Array without size (array with no subscript, or subscript is 0) is declared as a structure/union member.

239: Negative subscript

A negative value is specified as the size in an array type declaration.

240: Non-address expression

An address is required for an initialization expression. An error will occur in the examples below:

```
int a;  
int *b = a;
```

241: Non-constant initializer

The initializer is not a constant.

243: Null dimension

When defining a multi-dimensional array, other than at the first dimension, subscript values must be defined.

244: Prototype must have parameter types '<function>'

When declaring a function prototype, argument types must be specified using parameter type definition.

```
void func1( int arg1, int arg2 );    /* correct */  
void func2( arg1, arg2 );           /* incorrect */
```

245: Too many initializers

Number of initializers larger than the number of objects to be initialized is specified.

246: Zero size bit field '<identifier>'

0 is specified as the width of a named bit field.

<Undefined errors>

250: *Illegal struct/union name for member . '<member>'*

The left operand of a component selection operator is not structure or union type.

251: *Illegal struct/union pointer name for member ->'<member>'*

The left operand of a component selection operator is not a pointer to structure or union type.

252: *Undefined struct/union '<identifier>' of '<tag>'*

A structure/union with a tag name <tag> is not defined, therefore, <identifier> cannot be declared as a structure/union.

253: *Undefined struct/union '<identifier>', left of '<operator>'*

An undefined structure/union is used in the left operand expression of the component selection operator (">" or ".").

254: *Unknown size '<identifier>'*

Array without size was declared.

256: *'<identifier>' undefined*

An undefined identifier is used.

<Double definition errors>

260: *Duplicate case in switch '<value>'*

The same value is used twice for case labels in one switch statement.

261: *Duplicate default in switch*

Two or more default labels are written in one switch statement.

262: *Redeclaration of '<identifier>'*

An attempt was made to define an already-defined <identifier>.

263: *Redeclaration of struct/union/enum/tag '<tag>'*

An attempt was made to declare the tag name <tag> that had already been used in the struct, union, or enumeration declaration.

264: *Redeclaration of struct/union member '<member>'*

An attempt was made to declare members of the same name in the struct or union declaration.

<Operand errors>

270: *Bad left/right operand '<operator>'*

Operand type of the operator is illegal.

271: *Illegal cast*

An attempt was made to convert a type which cannot be converted. The type conversion in the example below is not allowed.

```
int a;
struct st{ int a1, b1; } st1;
st1 = (struct st)a;           /* incorrect */
```

272: *Illegal indirection*

Pointer operator "*" is used for a non-pointer value.

273: *Illegal sizeof*

The operand of sizeof operator must be either an object name or a type name.

274: *Illegal struct/union type, use '->'*

Component selection operator "." is used as a pointer to a member of a structure/union.

When specifying the member of a structure/union pointed to by a pointer, use operator ">".

275: *Illegal struct/union pointer type, use ' . '*

Component selection operator "." is used for a structure/union object. When specifying a member of a structure/union object, use operator ".".

276: *Illegal subscript*

Operator "[" is used for an object of other than array or pointer type.

277: *Unacceptable operand of '&'*

The operand of address operator "&" is illegal.

278: *'<member>' not member of struct/union*

The member name <member> is not a struct or union member.

<Expression errors>**280: *Cannot cast void to non-void***

Void type cannot be cast to another type.

281: *Illegal actual parameter '<value>'th of '<function>'*

At a function call, an argument has an error. The number <value> represents the argument number.

282: *Illegal cast to array type*

Object type is cast to array type.

283: *Illegal cast to function type*

Object type is cast to function type.

284: *Illegal compare struct/union*

Comparison between structures or unions is not allowed. Compare members of a structure or union.

285: *Illegal function*

A function is called using an identifier which is not declared as function type, or an expression which is not a pointer to a function.

286: *Illegal index, non-integral*

Value of an array subscript expression is not integer type.

288: *Illegal operand '<operator>'*

The operand is used incorrectly.

289: *Illegal operator '<operator>' for struct/union*

The structure or union cannot be operated on by the specified operator.

291: *Incompatible types '<identifier>'*

Operation on incompatible types is performed.

292: *Invalid addition, pointer to pointer*

Addition between pointer types is performed.

293: *Invalid addition/subtraction, pointer to non-integral value*

Addition or subtraction between pointer type and non-integer value is performed.

294: *Invalid subtraction pointer from non-pointer*

Attempt is made to subtract a pointer type value from a non-pointer type value.

295: *Lvalue required '<operator>'*

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

296: *Lvalue specifies const object*

Const type object value is used as a left-side value. Operation to change a const type object value is not allowed.

298: *Non-integral in switch expression*

The result of evaluating a switch expression is not an integer value.

300: Void type in expression

A void expression is used in a control expression of the if, while, for, or do statement.

301: '<operator>' needs lvalue

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

<Preprocessor errors>**310: Cannot open #include file "<filename>"**

The #include file <filename> cannot be found.

311: Illegal identifier '<identifier>' found in defined-operator

An unspecifiable <identifier> is used in the defined phrase of the #if or #elif statement.

313: Illegal macro name

An invalid identifier is specified as a macro name.

314: Illegal macro parameter '<parameter>'

An invalid character is specified as an argument in a function-type macro name definition.

316: Illegal #elif

The #elif is used incorrectly. The probable cause is that the corresponding #if, #ifdef, or #ifndef is missing.

317: Illegal #else

The #else is used incorrectly. The probable cause is that the corresponding #if, #ifdef, or #ifndef is missing.

318: Illegal #include filename

The file name specified in the #include command is illegal.

319: Illegal #line filename

The file name specified in directive #line is illegal.

320: Illegal #line number

The line number specified in directive #line is illegal.

322: Macro '<identifier>' redefined

Replacement contents differ in redefinition of the macro.

324: Too long token

The token name is excessively long.

325: Too many #else

Usage of #else is incorrect. You've already used #else in the corresponding to #if, #ifdef, or #ifndef.

326: Unexpected EOF in macro '<identifier>'

An EOF code is found in the macro call.

327: Unexpected EOF in '<preprocessing directive>'

An EOF code is found before the preprocessing directive corresponding to <preprocessing directive> is described.

328: Unexpected '<preprocessing directive>'

The preprocessing directive corresponding to <preprocessing directive> is missing.

329: Unknown preprocessor command

A incorrect preprocessing directive was used. Check the spelling of preprocessing directive.

330: #elif following #else

The #elif command is written after the #else command.

331: #error '<message>'

The message <message> is output by #error command.

- 332: ')' *not found in '<operator>'*
 ')' corresponding to '(' is missing in preprocessing operator.

<Limits exceeded>

- 340: *Too large bit field '<bit field>'*
 Number of bits larger than the standard number is specified at bit field declaration.
- 341: *Too many characters in constant*
 A number of characters exceeding the size of int type are specified in the character constant.
- 342: *Constant too big*
 Constant value exceeds the range in which a value can be expressed by the specified type.
- 343: *Out of range for enum constant*
 Enumeration constant exceeds the range in which a value can be expressed.

<Compiler limits exceeded>

- 360: *Compiler limit, too deep nesting of #if/#ifdef/#ifndef*
 The nesting level of #if, #ifdef, and #ifndef exceeded the compiler limit.
- 361: *Compiler limit, too deep nesting of #include*
 The nesting level of #include exceeded the compiler limit.
- 362: *Compiler limit, too large object in <memory_area> memory space*
 The object size exceeded the range which can be specified in <memory area>.
- 363: *Compiler limit, too many declarations*
 The number of qualifiers exceeds the compiler limit.
- 364: *Compiler limit, too many parameter*
 The number of arguments of a function exceeded the compiler limit.
- 365: *Compiler limit, too many -I options*
 The number of specifications of option -I exceeded the compiler limit.
- 366: *Compiler limit, too many -D options*
 The number of specifications of option -D exceeded the compiler limit.
- 367: *Compiler limit, too many -U options*
 The number of specifications of option -U exceeded the compiler limit.

<Errors related to extension function>

- 370: *Bad inline assemble construction*
 Syntax for inline assembly is illegal.
- 371: *Duplicate function attribute*
 Two or more function qualifiers (eg, __cdecl) were specified within one declaration.
- 372: *Illegal function call, function defined __interrupt or __regbank*
 __interrupt type function cannot be called from a C program.
- 373: *Illegal function return type*
 __interrupt type function must be void.
- 374: *Illegal parameter type lists*
 __interrupt type function cannot specify parameters.
- 375: *Illegal pointer type, different function attribute*
 The pointer type which differ in function type is used.
- 385: *Illegal function type for inline/builtin-function '<function>'*
 The specified function cannot be made into the inline function.

<Others>

390: *division/remainder by zero*

Divisor 0 or remainder 0 occurred during evaluation of a constant expression.

<Errors of preprocessor>

400: *Illegal pointer size*

The pointer size specification is illegal.

401: *Illegal displacement size*

The displacement size specification is illegal.

402: *Illegal section size*

The section size specification is illegal.

403: *Illegal assignment of pseudo register-variable '<register-name>'*

The incorrect value is assigned to a pseudo register variable.

404: *Duplicate memory size*

Two or more displacement are included.

406: *Cannot initialize to variable*

An initial value cannot be specified for an io variable.

407: *Cannot declare io variable in block-scoped*

An io variable cannot be specified within a block.

408: *Not support keyword '___<keyword>'*

The specified extended reserved word is not supported.

410: *Illegal pointer operation '<operation>', pointer is different attribute*

The different pointer type are specified.

12.3 C Compiler Warning

The number in parenthesis which follows to error messages shows a warning level.

- | | |
|-------------|------------------------------|
| (1) Level 1 | Normal warning (Default -w1) |
| (2) Level 2 | Middle warning (-w2) |
| (3) Level 3 | Detail warning (-w3) |

In warning level 3, when a program is not exactly written, the warning message is output. The description which may cause a mistake can be checked. Note what the message meaning says, because the same message is shown with different levels according to circumstances.

<Warnings of option>

500: *Duplicate option '<option>' (1)*

The same option is specified twice or more.

501: *Illegal option '<option>' (1)*

Specification of the option is illegal. This option is ignored.

503: *Option '<option>' requires an argument (1)*

The parameter is not specified for the option which necessitates the parameter. This option is ignored.

<Warnings of discarding data>**510: *Floating-point overflow* (2)**

A floating point operation exception such as overflow or underflow occurred during a floating constant operation. The compiler continues the operation, regarding the operation result as 0.0.

511: *Out of range in hex escape sequence '<Hexadecimal_constant>'* (1)

The number of digits in the hexadecimal escape sequence of a character string literal exceeded 9 characters.

513: *Too many initialize for array '<identifier>'* (1)

The initializers more than the number of initializing objects are specified. The excess initializers are ignored.

514: *Type conversion, possible loss of data* (2)

Because of values of different base types were specified in one expression, the type of one of them was converted. The data were discarded during type conversion.

515: *Too long identifier, truncated to '<identifier>'* (1)

The identifier was too long; characters following the 1023rd character were discarded. Until 1023rd characters, the same characters may be regarded as the same identifier.

516: *Integral overflow* (2)

The operation result exceeds the range which can be represented by integer.

517: *Too big for character* (1)

The internal code of the character constant exceeds the range which can be handled.

<Warnings of undefine>**521: *Undefined return type in '<function>'* (3)**

A function which is not declared or defined is used. The compiler continues the operation, regarding the function as one which returns an int type value.

522: *Unnamed struct/union as parameter* (3)

A tag name is not declared for a structure/union is used in an argument for a function call.

523: *Undefined struct/union '<tag-name>'* (3)

An undefined structure/union is used. They are compiled as a structure/union without members.

<Warnings of unnecessary declaration/description>**530: *Duplicate type qualifier* (1)**

The same qualifier (const or volatile) was used twice or more.

531: *No identifier declared* (2)

A identifier is not declared, so it was ignored.

532: *Untagged enum/struct/union declared no symbols* (2)

A identifier of a enumeration/structure/union without tag is not declared, so this declaration is ignored.

534: *'<type>' may be used on integral types only* (1)

Signed or unsigned is used for a type other than integer type. The specification is ignored.

536: *No use memory size* (1)

It is not able to specify the displacement or I/O to a automatic variables.

537: *Label '<label>' defined but not used in function* (3)

A label which is not used is included.

- 538: *Statement not reached* (1)**
 A statement which is not executed is included.

<Descriptions which might cause errors>

- 540: *Assignment in conditional expression* (3)**
 The result of an assignment expression is used as a conditional expression.
 For example, this warning is output in the following case:

```
if (result = func(1))
```

- 541: *Case constant '<value>' too big for the type of switch expression* (1)**
 Case label value exceeded the range in which values can be represented by the conditional expression type in the switch statement.
- 542: *Comma operator in array index expression* (3)**
 Comma operator is used in an array subscript expression.
- 543: *Constant in conditional expression* (3)**
 A constant is specified in an if or while conditional expression. This warning is just for information.
- 544: *Const object '<object>' should be initialized* (3)**
 A const object is not initialized.
- 545: *Illegal assignment, const/volatile qualifier mismatch* (1)**
 A pointer to an object which was declared as const or volatile is assigned to a pointer to an object which was not declared as const or volatile.
- 546: *Illegal conversion, integral type mismatch* (3)**
 During conversion of two integer values, data are lost. For example, this warning is output in the following case:

```
short v_short;
long  v_long;
v_short = v_long;
```

- 547: *Illegal conversion, floating type mismatch* (3)**
 During conversion of two floating point values, data are lost. For example, this warning is output in the following case:

```
float  v_float;
double v_double;
v_float = v_double;
```

- 548: *Illegal pointer operation '<operator>', array's subscript mismatch* (1)**
 A pointer operation to an array which have different subscript is performed.
- 549: *Illegal pointer operation '<operator>', indirection level mismatch* (1)**
 The level of indirect reference is not consist.

```
char **pptr;
char *ptr;
ptr = pptr; /* warning */
```

- 550: *Illegal pointer operation '<operator>', type mismatch* (1)**
 Pointer to different types is used in a expression.

```
char *cp;  
int *ip;  
cp = ip;    /* warning */
```

551: *Logical operation '<operator>' on address of string constant* (3)

A logical operation using a string literal address is performed. For example, this warning is output in the following example:

```
char *str = "Hello";  
if (str == "Hello")
```

552: *Meaningless statement* (2)

Meaningless statement is written.

553: *No return value in '<function>'* (2)

A return statement is not written in a function which is declared for a return value.

556: *'<identifier>' used before set in '<function>'* (3)

A variable which has had no value set is referenced.

557: *Redeclaration of '<identifier>', array's subscript mismatch* (1)

Different array elements are declared.

558: *Multiple comparison operator in expression* (3)

Comparison operations are written twice or more.

559: *Out of range for array* (3)

Accessing is executed out of the range for the specified array.

<Warnings to specifications>

560: *Bad storage class '<identifier>'* (1)

A storage-class specifier which cannot be used is specified.

561: *Cannot return value for void function* (1)

A return statement is written within a function declared as void, which does not return a value.

563: *Storage-class specifier after type* (3)

A Storage-class specifier is declared after a type specifier in the declaration.

565: *Redeclaration of '<function>', class mismatch* (3)

A function is redefined in different storage-class.

<Warnings for prototype>

571: *Illegal declaration with formal argument list* (1)

Arguments are not declared for definition of a function which was declared as using arguments. In subsequent function calls, the function is regarded as not using arguments.

572: *Illegal assignment of function's pointer, different parameter lists* (1)

A pointer to a function with different argument type or number is assigned to another function pointer.

573: *Illegal declaration without formal argument list* (1)

Parameters are declared in the definition of a function which was declared as not using arguments (void). After the definition, the function is regarded as having arguments defined.

574: *Illegal function call '<function>', declared with void* (1)

The argument are specified at calling function which is declared without argument(void).

- 577: *Illegal function call '<function>', too few actual parameters (1)***
The number of arguments specified for a function call is smaller than the number of parameters declared at function declaration or definition.
- 578: *Illegal function call '<function>', too many actual parameters (1)***
The number of arguments specified for a function call is larger than the number of parameters declared at function declaration or definition.
- 579: *Illegal function call '<function>', type conversion (1)***
The size of argument type differ from that of parameter type. The type of actual arguments is converted into that of parameters.
- 580: *Illegal prototype in '<number >th parameter (1)***
The same function is declared twice or more but the parameter list types do not match.
- 581: *Different declaration parameter list from definition (1)***
The type of the parameter list at function declaration differs from the type of the parameter list at function definition. The parameter list at function definition is used.
- 582: *No function prototype '<function>' (2)***
A function whose prototype is not declared is called.
- 583: *Parameter type mismatch, '<value>th parameter of '<function>' (1)***
An argument of a different type was passed instead of the parameter specified at function definition.
- 584: *Parameter number mismatch in prototype (1)***
The same function is declared twice or more. The number of parameters is different for each declaration.
- 585: *Uses old-style declarator '<function>' (3)***
Function declaration and definition are old-style.
- 586: *No use in formal-parameter list '<identifier>' (1)***
When function definition is old-style, a type which is not used is defined.
- 587: *Illegal function's pointer type, different return type (1)***
The function pointer is used erroneously because the type of return value is incorrect.

<Warnings of preprocessor>

- 591: *Illegal/missing macro name (1)***
An invalid macro name is specified or the macro name is not specified.
- 592: *Illegal macro call '<macro name>', mismatched number of parameters (1)***
The number of arguments in a function format macro call does not match the number of arguments in the macro definition.
- 596: *Illegal ##, beginning of a macro definition (1)***
The macro definition replacement list starts with a ## operator.
- 597: *Illegal ##, ending of a macro definition (1)***
The macro definition replacement list ends with a ## operator.
- 598: *Macro formal parameter expected after # (1)***
The operand after a # operator in a macro definition must be a parameter name.
- 599: *Unexpected character after directive, ignored (1)***
An invalid character string is specified after a preprocessing directive. The character string is ignored.

<Warnings of pragma>

- 610: *#pragma keyword expected, '<token>' found, (1)***
 <token> after #pragma was not identified as a command. This directive is ignored.
- 611: *#pragma [on \ off] expected (1)***
 #pragma directive needs an on or off parameter, but the parameter was not specified or the specified parameter was not identified. This directive is ignored.
- 612: *#pragma [1 \ 2 \ 4] expected (1)***
 #pragma directive needs a 1, 2, or 4 parameter, but the specified parameter was not identified. This directive is ignored.
- 615: *Unexpected #pragma token '<token>' (1)***
 An unnecessary token was found in the argument list of #pragma directive. The remaining part of this directive is ignored.
- 616: *Cannot use #pragma disinterrupt for inline/builtin-function (1)***
 #pragma disinterrupt is specified for a inline function. Since this function expands code directly, disinterrupt cannot be specified for it. This #pragma directive will be ignored.
- 618: *Unknown #pragma (1)***
 #pragma directive not supported by the compiler is used. This directive is ignored.
- 619: *Cannot use #pragma in initializing (1)***
 #pragma directive is used in the middle of initializing.

<Others>

- 620: *Cannot use function attribute '<identifier>' (1)***
 Specified function qualifier <identifier> cannot be specified. The function qualifier is ignored.
- 621: *Illegal escape sequence '<character>' (1)***
 An invalid character is specified for an escape sequence after the escape character. The escape character is ignored.
- 622: *Sizeof returns 0 (1)***
 The size of the operand of the sizeof operator is 0.
- 623: *Type definition in formal parameter list '<tag>' (1)***
- 624: *Type definition in formal parameter list (no tag) (1)***
 Structure/union/enumeration type is declared in a formal parameter list. This declaration is regarded as an external declaration. If untagged structure/union/enumeration type was declared, the tag name is indicated as "no tag".
- 626: **/' found outside of commnet (1)***
 "*/" is written outside a comment. For example, the warning is output in the following case:

```
int /* comment */ptr;
```


 After processing the above, the following is regarded as:

```
int *ptr;
```
- 631: *'<character>' found inside of commnet (3)***
 "/*" and "/*" are found in a comment. This description is ignored.
- 632: *Static function '<function>' not found (1)***
 A static function which has no entities is found. If a static function is declared, but not defined, this warning is generated.
- 634: *Too large function '<function_name>', optimize not performed (2)***
 Several optimization cannot be performed because the function is too large.

635: *Too much register pseudovvariable use, value may be invalid* (1)

The value may be invalid because of the pseudo-variables are used too much.

636: *Illegal value of assignment for pseudo register-variable '<function_name>'* (1)

An invalid value is assigned to the pseudo-variable.

<Warnings of processor dependent extension function>

651: *Illegal displacement size* (1)

A displacement, which cannot be used, is specified.

652: *Illegal section size* (1)

A displacement which was not able to be used was specified for the section.

653: *Illegal pointer operation '<operator>', pointer size mismatch* (1)

The operation which doesn't suit the size of the pointer is performed.

APPENDIX

A ANSI Processing System Dependence Specifications

This chapter describes C language implementation-defined behaviors set forth in ISO 9899 : 1990 ANSI.

The operation of the implementation-defined is an operation that depends on the processing system characteristics of the operations relating to correct program structure elements and correct data, and it is also an operation for which each processing system must be documented. Also, the processing system runs in a specific translation environment under specific control options, translates programs for specific execution environments, and is stipulated as a specific software collection that supports the execution of functions for that execution environment.

The operation of the implementation-defined noted in this chapter is as follows, and the JIS item number is shown in the detailed description of each item.

3.4	Byte
3.14	Object
5.1.1.2	Translation phases
5.1.2.1	Freestanding environment
5.1.2.2.1	Program startup
5.1.2.3	Program execution
5.2.1	Character sets
5.2.4.2.1	Sizes of integral types
5.2.4.2.2	Characteristics of floating types
6.1.2.5	Types
6.1.3.1	Floating constants
6.1.3.4	Character constants
6.1.7	Header names
6.2.1.1	Characters and integers
6.2.1.2	Signed and unsigned integers
6.2.1.3	Floating and integral
6.2.1.4	Floating types
6.3.2.3	Structure and union members
6.3.3.4	The sizeof operator
6.3.4	Cast operators
6.3.5	Multiplicative operators
6.3.7	Bitwise shift operators
6.5.1	Storage class specifier
6.5.2.1	Structure specifier and union specifier
6.5.3	Type specifiers
6.8.1	Conditional inclusion
6.8.2	Source file inclusion
6.8.6	Pragma directive
6.8.8	Predefined macro names

A.1 Definitions and Conventions

A.1.1 [3.4 Byte]

ISO/ANSI C

A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined.

TLCS-900 C operation

One byte consists of 8 bits.

A.1.2 [3.14 Object]

ISO/ANSI C

Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.

TLCS-900 C operation

Explicitly specified. Please refer to "B Translation Limits".

A.2 Environment

A.2.1 [5.1.1.2 Translation phases]

ISO/ANSI C

Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

TLCS-900 C operation

Nonempty sequence of white-space characters other than new-line is replaced by one space character.

A.2.2 [5.1.2.1 Freestanding environment]

Explanation

In a freestanding environment in which C program execution may take place without any benefit of an operating system.

ISO/ANSI C

The name and type of the function called at program startup are implementation-defined.

TLCS-900 C operation

The name of the function called at program startup is main, and type is user-defined.

ISO/ANSI C

Any library facilities available to a freestanding program are implementation-defined.

TLCS-900 C operation

Usable libraries are detailed in the Part 6 "Standard Library Functions".

ISO/ANSI C

The effect of program termination in a freestanding environment is implementation-defined.

TLCS-900 C operation

The processing at program termination is user-defined.

A.2.3 [5.1.2.2.1 Program startup]**ISO/ANSI C**

If the value of **argc** is greater than zero, the array members **argv[0]** through **argv[argc-1]** inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup.

TLCS-900 C operation

This is irrelevant to the TLCS-900, because it does not operate in the host environment.

A.2.4 [5.1.2.3 Program execution]**ISO/ANSI C**

What constitutes an interactive device is implementation-defined.

TLCS-900 C operation

Interactive devices can be defined by the user.

Explanation

Interactive device : Refers to an input device such as the keyboard and an output device such as the display unit.

ISO/ANSI C

More stringent correspondences between abstract and actual semantics may be defined by each implementation.

TLCS-900 C operation

Currently, abstract and actual semantics are not strictly corresponded to each other. For reasons of optimization effect, abstract and actual semantics do not corresponded one for one.

A.2.5 [5.2.1 Character sets]**ISO/ANSI C**

The values of the members of the execution character set are implementation-defined.

TLCS-900 C operation

The values of the members of the execution character set conform to ASCII code.

A.2.6 [5.2.4.2.1 Sizes of integral types]**ISO/ANSI C**

Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

: number of bits for smallest object that is not a bit-field (byte)	
CHAR_BIT	8
: minimum value for an object of type signed char	
SCHAR_MIN	-127
: maximum value for an object of type signed char	
SCHAR_MAX	+127
: maximum value for an object of type unsigned char	
UCHAR_MAX	255
: minimum value for an object of type char	
CHAR_MIN	<i>see below</i>
: maximum value for an object of type char	
CHAR_MAX	<i>see below</i>
: maximum number of bytes in a multibyte character, for any supported locale	
MB_LEN_MAX	1
: minimum value for an object of type short int	
SHRT_MIN	-32767
: maximum value for an object of type short int	
SHRT_MAX	+32767
: maximum value for an object of type unsigned short int	
USHRT_MAX	65535
: minimum value for an object of type int	
INT_MIN	-32767
: maximum value for an object of type int	
INT_MAX	+32767
: maximum value for an object of type unsigned int	
UINT_MAX	65535
: minimum value for an object of type long int	
LONG_MIN	-2147483647
: maximum value for an object of type long int	
LONG_MAX	+2147483647
: maximum value for an object of type unsigned long int	
ULONG_MAX	4294967295

If the value of an object of type char is treated as a signed integer when used in an expression, the value of CHAR_MIN shall be the same as that of SCHAR_MIN and the value of CHAR_MAX shall be the same as that of SCHAR_MAX. Otherwise, the value of CHAR_MIN shall be 0 and the value of CHAR_MAX shall be the same as that of UCHAR_MAX. (See [6.1.2.5 Types])

TLCS-900 C operation

CHAR_BIT	8	
SCHAR_MIN	-128	
SCHAR_MAX	+127	
UCHAR_MAX	255	
CHAR_MIN	-128 as default	(0 with -Xub option)
CHAR_MAX	+127 as default	(255 with -Xub option)
MB_LEN_MAX	4	

SHRT_MIN	-32768
SHRT_MAX	+32767
USHRT_MAX	65535
INT_MIN	-32768
INT_MAX	+32767
UINT_MAX	65535
LONG_MIN	-2147483648
LONG_MAX	+2147483647
ULONG_MAX	4294967295

A.2.7 [5.2.4.2.2 Characteristics of floating types]

ISO/ANSI C

The rounding mode for floating-point addition is characterized by the value of FLT_ROUNDS:

- 1 indeterminate
- 0 toward zero
- 1 to nearest
- 2 toward positive infinity
- 3 toward negative infinity

All other values for FLT_ROUNDS characterize implementation-defined rounding behavior.

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal or greater in magnitude (absolute value) to those shown, with the same sign:

- : radix of exponent representation, b
 - FLT_RADIX 2
- : number of base-FLT_RADIX digits in the floating-point significand, p
 - FLT_MANT_DIG
 - DBL_MANT_DIG
 - LDBL_MANT_DIG
- : number of decimal digits, q , such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix b digits and back again without change to the q decimal digits, $[(p-1) \times \log_{10} b] + \begin{cases} 1 & \text{if } b \text{ is a power of } 10 \\ 0 & \text{otherwise} \end{cases}$
 - FLT_DIG 6
 - DBL_DIG 10
 - LDBL_DIG 10
- : minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number, e_{\min}
 - FLT_MIN_EXP
 - DBL_MIN_EXP
 - LDBL_MIN_EXP
- : minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $[\log_{10} b^{e_{\min}-1}]$
 - FLT_MIN_10_EXP -37
 - DBL_MIN_10_EXP -37
 - LDBL_MIN_10_EXP -37
- : maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number, e_{\max}

FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP

: maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $[\log_{10} ((1-b^{-p}) \times b^{e_{\max}})]$

FLT_MAX_10_EXP +37
DBL_MAX_10_EXP +37
LDBL_MAX_10_EXP +37

The values given in the following list shall be replaced by implementation-defined expressions with values that shall be greater than or equal to or those shown:

: maximum representable finite floating-point number, $(1-b^{-p}) \times b^{e_{\max}}$

FLT_MAX 1E+37
DBL_MAX 1E+37
LDBL_MAX 1E+37

The values given in the following list shall be replaced by implementation-defined expressions with values that shall be less than or equal to those shown:

: the difference between 1 and the least value greater than 1 that is representable in the given floating-point type, **Error! Bookmark not defined.** b^{1-p}

FLT_EPSILON 1E-5
DBL_EPSILON 1E-9
LDBL_EPSILON 1E-9

: minimum normalized positive floating-point number, **Error! Bookmark not defined.**

Error! Bookmark not defined. $b^{e_{\min}-1}$

FLT_MIN 1E-37
DBL_MIN 1E-37
LDBL_MIN 1E-37

TLCS-900 C operation

FLT_ROUNDS 1
FLT_RADIX 2
FLT_MANT_DIG 24
DBL_MANT_DIG 53
LDBL_MANT_DIG 64
FLT_DIG 6
DBL_DIG 15
LDBL_DIG 18
FLT_MIN_EXP (-125)
DBL_MIN_EXP (-1021)
LDBL_MIN_EXP (-16381)
FLT_MIN_10_EXP (-37)
DBL_MIN_10_EXP (-307)
LDBL_MIN_10_EXP (-4931)
FLT_MAX_EXP (+128)
DBL_MAX_EXP (+1024)

LDBL_MAX_EXP	(+16384)
FLT_MAX_10_EXP	(+38)
DBL_MAX_10_EXP	(+308)
LDBL_MAX_10_EXP	(+4932)
FLT_MAX	3.402823466e+38F
DBL_MAX	1.7976931348623157e+308
LDBL_MAX	1.7976931348623157e+308
FLT_EPSILON	1.192092896e-07F
DBL_EPSILON	2.2204460492503131e-16
LDBL_EPSILON	1.084202172485504434e-019L
FLT_MIN	1.175494351e-38F
DBL_MIN	2.2250738585072014e-308
LDBL_MIN	2.2250738585072014e-308

A.3 Language

A.3.1 [6.1.2.5 Types]

ISO/ANSI C

An object declared as type **char** is large enough to store any member of the basic execution character set. If a member of the required source character set enumerated in 5.2.1 is stored in a **char** object, its value is guaranteed to be positive. If other quantities are stored in a **char** object, the behavior is implementation-defined: the values are treated as either signed or nonnegative integers.

TLCS-900 C operation

The values are treated as signed integers.

A.3.2 [6.1.3.1 Floating constants]

ISO/ANSI C

If the scaled value is in the range of representable values (for its type) the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner.

TLCS-900 C operation

The "nearest representable value" is returned.

Explanation

How the multiplication result of floating-point numbers is rounded is implementation-defined.

A.3.3 [6.1.3.4 Character constants]

ISO/ANSI C

An integer character constant is a sequence of one or more multibyte characters enclosed in single-quotes, as in **'x'** or **'ab'**. A wide character constant is the same, except prefixed by the letter **L**. With a few exceptions detailed later, the elements of the sequence are any members of the source character set; they are mapped in an implementation-defined manner to members of the execution character set.

TLCS-900 C operation

Conforms to ASCII code. TLCS-900 family C compiler can not treat a wide character constant.

ISO/ANSI C

The value of an integer character constant containing more than one character, or containing a character or escape sequence not represented in the basic execution character set, is implementation-defined.

TLCS-900 C operation

Conforms to ASCII code. A character constant can hold to 4 bytes.

ISO/ANSI C

The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

TLCS-900 C operation

TLCS-900 family C compiler can not treat a wide character constant.

A.3.4 [6.1.7 Header names]**ISO/ANSI C**

The sequences in both forms of header names are mapped in an implementation-defined manner to headers or external source file names as specified in 6.8.2.

TLCS-900 C operation

Refer to A.3.18 [6.8.2 Source file inclusion].

A.3.5 [6.2.1.1 Characters and integers]**ISO/ANSI C**

Whether a "plain" **char** is treated as signed is implementation-defined.

TLCS-900 C operation

Treated as signed. It can change by -Xuc option.

A.3.6 [6.2.1.2 Signed and unsigned integers]**ISO/ANSI C**

When a value with integral type is demoted to a signed integer with smaller size, or an unsigned integer is converted to its corresponding signed integer, if the value cannot be represented the result is implementation-defined.

TLCS-900 C operation

When the converted value cannot be represented, it is converted to an unsigned integer of the same size as the signed integer, which is then sign-extended to obtain the result.

A.3.7 [6.2.1.3 Floating and integral]**ISO/ANSI C**

When a value of integral type is converted to floating type, if the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower value, chosen in an implementation-defined manner.

TLCS-900 C operation

It is rounded up or down depending on the bit one bit below the effective significand part; it is raised to a unit when 1 or discarded when 0. This process is performed by floating-point runtime library.

A.3.8 [6.2.1.4 Floating types]**ISO/ANSI C**

When a **double** is demoted to **float** or a **long double** to **double** or **float**, if the value being converted is outside the range of values that can be represented, the behavior is undefined. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower value, chosen in an implementation-defined manner.

TLCS-900 C operation

Same as described in A.3.7 [6.2.1.3 Floating and integral].

A.3.9 [6.3.2.3 Structure and union members]**ISO/ANSI C**

With one exception, if a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation-defined. One special guarantee is made in order to simplify the use of unions: If a union contains several structures that share a common initial sequence, and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them.

TLCS-900 C operation

In TLCS-900, the byte ordering in memory is little endian.

A.3.10 [6.3.3.4 The sizeof operator]**ISO/ANSI C**

The value of the result is implementation-defined, and its type (an unsigned integral type) is **size_t** defined in the `<stddef.h>` header.

TLCS-900 C operation

The `size_t` is an unsigned long type.

char	1
short	2
int	2
long	4
float	4
double	8
long double	10

pointer 4

A.3.11 [6.3.4 Cast operators]

ISO/ANSI C

A pointer may be converted to an integral type. The size of integer required and the result are implementation-defined.

An arbitrary integer may be converted to a pointer. The result is implementation-defined.

TLCS-900 C operation

A value of a pointer is treated as a unsigned long type. Moreover, if it is a value within the limits which can be treated with unsigned long type, it is possible to use it which is converted to a pointer.

A.3.12 [6.3.5 Multiplicative operators]

ISO/ANSI C

If either operand is negative, whether the result of the `/` operator is the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient is implementation-defined, as is the sign of the result of the `%` operator.

TLCS-900 C operation

If either operand is negative, the result of the `/` operator is the smallest integer greater than or equal to the algebraic quotient. The sign of the result of the `%` operator is

If a dividend of the `%` operator is negative value, the sign of the result is negative, the other is positive.

A.3.13 [6.3.7 Bitwise shift operators]

ISO/ANSI C

The result of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of `E1` divided by the quantity, 2 raised to the power `E2`. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

TLCS-900 C operation

If `E1` is a negative value, the result of `E1 >> E2` is that of an arithmetic shift-right operation. Namely, the sign is retained.

A.3.14 [6.5.1 Storage-class specifiers]

ISO/ANSI C

A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.

TLCS-900 C operation

Objects declared using storage-class specifier **register** are treated in the same as those that are specified with storage-class specifier **auto**.

A.3.15 [6.5.2.1 Structure and union members]**ISO/ANSI C**

Whether the high-order bit position of a (possibly qualified) "plain" **int** bit-field is treated as a sign bit is implementation-defined.

TLCS-900 C operation

Treated as a sign bit.

ISO/ANSI C

An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined.

TLCS-900 C operation

Bit-fields does not overlaps adjacent units. The order of allocation of bit-fields within a unit is allocated from high-order(MSB). The order of allocation can change by -Xw option.

ISO/ANSI C

Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

TLCS-900 C operation

Refer to "TLCS-900 Compiler System User's Guide" for the alignment of each non-bit-field member of a structure or union object.

A.3.16 [6.5.3 Type specifiers]**ISO/ANSI C**

What constitutes an access to an object that has volatile-qualified type is implementation-defined.

TLCS-900 C operation

An access processing to an object that has volatile-qualified type is not deleted due to optimization or changed order, except when allowed under rules for evaluation of expressions.

A.3.17 [6.8.1 Conditional inclusion]**ISO/ANSI C**

Whether the numeric value for character constants matches the value obtained when an identical character constant occurs in an expression (other than within a **#if** or **#elif** directive) is implementation-defined. Also, whether a single-character character constant may have a negative value is implementation-defined.

TLCS-900 C operation

The numeric value for character constants matches the value obtained when an identical character constant occurs in an expression (other than within a `#if` or `#elif` directive). Also, a single-character character constant can have a negative value.

A.3.18 [6.8.2 Source file inclusion]**ISO/ANSI C**

A preprocessing directive of the form

include *<h-char-sequence>* *new-line*

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the `<` and `>` delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

TLCS-900 C operation

The search order of the header is as follows.

1. the directory indicated by `-I` option
2. include directory under the directory indicated by environment variable `THOME900`

ISO/ANSI C

A preprocessing directive of the form

include *"q-char-sequence"* *new-line*

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the `"` delimiters. The named source file is searched for in an implementation-defined manner.

TLCS-900 C operation

The search order of the source file is as follows.

1. the directory of the source file during the compiling processing.
2. the directory indicated by `-I` option
3. include directory under the directory indicated by environment variable `THOME900`

ISO/ANSI C

The method by which a sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined.

TLCS-900 C operation

The usable character in header name(contain path name) is as follows.

alphabet and number

space character

! # \$ % & ' () - @ _ { } ~

The nonusable character in header name(contain path name) is as follows.

except for ASCII code 0x20 - 0x7E

+ , ; = []

ISO/ANSI C

There shall be implementation-defined mapping between the delimited sequence and the external source file name.

TLCS-900 C operation

It is mapped by generating a path for the external source file name and passing it to the OS.

A.3.19 [6.8.6 Pragma directive]

ISO/ANSI C

A preprocessing directive of the form

#pragma *pp-tokens_{opt} new-line*

causes the implementation to behave in an implementation-defined manner.

TLCS-900 C operation

Refers to "3.3 #pragma Directives".

A.3.20 [6.8.8 Predefined macro names]

ISO/ANSI C

__DATE__

If the date of translation is not available, an implementation-defined valid date shall be supplied.

__TIME__

If the time of translation is not available, an implementation-defined valid time shall be supplied.

TLCS-900 C operation

The compiler stops compiling unless the correct date and time are obtained. However, the compiler does not determine whether its own data and time are correct.

B Translation Limits

This chapter describes the processing system translation limits that is set forth in ISO 9899 : 1990 ANSI.

The limits value of TLCS-900 C processing shows in parenthesis " () ",and the limits value of the above standard shows in square bracket " [] ".

- ☐ Nesting levels of blocks (15)[15]
- ☐ Nesting levels of conditional inclusion (255)[8]
- ☐ Pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete type in a declaration (12)[12]
- ☐ Nesting levels of parenthesized declarators within a full declarator (31)[31]
- ☐ Nesting levels of parenthesized expressions within a full expression (32)[32]
- ☐ Significant initial characters in an internal identifier or a macro name (1023)[31]
- ☐ Significant initial characters in an external identifier (1023)[6]
- ☐ External identifiers in one translation unit (512)[511]
- ☐ Identifiers with block scope declared in one block (no limit)[127]
- ☐ Macro identifiers simultaneously defined in one preprocessing translation unit (8192)[1024]
- ☐ Parameters in one function definition (31)[31]
- ☐ Arguments in one function call (31)[31]
- ☐ Parameters in one macro definition (31)[31]
- ☐ Arguments in one macro invocation (31)[31]
- ☐ Characters in a logical source line (10000)[509]
- ☐ Characters in a character string literal or wide string literal (10000)[509]
- ☐ Nesting levels for #included files (255)[8]
- ☐ Case labels for a switch statement (excluding those for any nested switch statements) (no limit)[257]
- ☐ Members in a single structure or union (no limit)[127]
- ☐ Enumeration constants in a single enumeration (65535)[127]
- ☐ Levels of nested structure or union definitions in a single struct-declaration-list (15)[15]

The following describes the internal compiler the processing system translation limits that is not set forth in ISO 9899 : 1990 ANSI.

- ☐ Number of lines of one source file that can be compiled (including line feed characters) (65535)
- ☐ Number of lines of one assembler file for which compiling results are output (65535)
- ☐ Number of characters that can be described in inline assembly (1024)
- ☐ Number of times that -I option can be specified (31)

History

Issue	Date	Update
1st Edition	7 Jan, 2009	1st Edition

TLCS-900 C Compiler Reference [1st Edition]

The Date of Issue: 7 Jan, 2009

TDE121-01