Table of Contents

[1. Language Specification 5](#_Toc9605428)

[1.1 Basic language specification 5](#_Toc9605429)

[1.1.1 Conforming language specification 5](#_Toc9605430)

[1.1.2 Constraints on the conforming language specification 5](#_Toc9605431)

[1.1.2.1 main function 5](#_Toc9605432)

[1.1.2.2 Constraints on types 5](#_Toc9605433)

[1.1.2.3 Constraints on keywords 5](#_Toc9605434)

[1.1.2.4 Constraints on identifiers 5](#_Toc9605435)

[1.1.2.5 Constraints on operations 5](#_Toc9605436)

[1.1.2.6 Constraints on pointer types 9](#_Toc9605437)

[1.1.2.7 Constraints on constants 9](#_Toc9605438)

[1.1.2.8 Constraints on string literals 9](#_Toc9605439)

[1.1.2.9 Constraints on compound literals 9](#_Toc9605440)

[1.1.2.10 Constraints on address constants 9](#_Toc9605441)

[1.1.2.11 Constraints on data and function declarations 9](#_Toc9605442)

[1.1.2.12 Constraints on preprocessor directives 9](#_Toc9605443)

[1.1.2.13 Notes on the initial value of a global variable 10](#_Toc9605444)

[1.1.2.14 Instruction output specification for a right shift operation 10](#_Toc9605445)

[1.1.2.15 Restrictions on library functions 10](#_Toc9605446)

[1.1.3 Floating-point operations 10](#_Toc9605447)

[1.1.3.1 Constraints on rounding mode 10](#_Toc9605448)

[1.1.4 Restrictions on formulas 10](#_Toc9605449)

[1.2 Runtime environment 11](#_Toc9605450)

[1.2.1 Internal representation of data 11](#_Toc9605451)

[1.2.1.1 Integer types 11](#_Toc9605452)

[1.2.1.2 Floating-point type 11](#_Toc9605453)

[1.2.1.3 Pointer types 11](#_Toc9605454)

[1.2.1.4 Enumerated types 11](#_Toc9605455)

[1.2.1.5 Array types 12](#_Toc9605456)

[1.2.1.6 Structure types 12](#_Toc9605457)

[1.2.1.7 Union types 12](#_Toc9605458)

[1.2.1.8 Bit fields 12](#_Toc9605459)

[1.2.1.9 Alignment conditions 12](#_Toc9605460)

[1.2.2 Sections 12](#_Toc9605461)

[1.2.2.1 List of sections 12](#_Toc9605462)

[1.2.2.2 Notes on memory access to lwm and lwm\_comm 13](#_Toc9605463)

[1.2.3 Function call interface 14](#_Toc9605464)

[1.2.3.1 Using the Processor registers 14](#_Toc9605465)

[1.2.3.2 Rules for setting for arguments and using parameters 14](#_Toc9605466)

[1.2.3.3 Rules for setting and using a return value 15](#_Toc9605467)

[1.2.3.4 Rules for built-in function \_\_getX and \_\_getY 15](#_Toc9605468)

[1.2.3.5 Interface between functions for R8 to R15 16](#_Toc9605469)

[1.2.3.6 Rules for stacks 16](#_Toc9605470)

[1.2.3.7 External names 17](#_Toc9605471)

[1.3 Extended language specification 18](#_Toc9605472)

[1.3.1 Vector types 18](#_Toc9605473)

[1.3.1.1 Vector types that can be used 18](#_Toc9605474)

[1.3.1.2 Loading or storing vectors 18](#_Toc9605475)

[1.3.1.3 Reading and writing vector elements 20](#_Toc9605476)

[1.3.1.4 Constraints on vectors 20](#_Toc9605477)

[1.3.1.5 Restrictions on writing vector elements 21](#_Toc9605478)

[1.3.2 #pragma 22](#_Toc9605479)

[1.3.2.1 #pragma inline, #pragma noinline 22](#_Toc9605480)

[1.3.2.2 #pragma inline\_asm 23](#_Toc9605481)

[1.3.2.3 #pragma stacksize 27](#_Toc9605482)

[1.3.2.4 #pragma entry 27](#_Toc9605483)

[1.3.2.5 #pragma section 28](#_Toc9605484)

[1.3.2.6 #pragma unroll 28](#_Toc9605485)

[1.3.2.7 #pragma gwm\_write\_only 29](#_Toc9605486)

[1.3.3 Built-in functions 30](#_Toc9605487)

[1.3.3.1 List of built-in functions 30](#_Toc9605488)

[1.3.3.2 Notes on describing built-in functions 41](#_Toc9605489)

[1.4 Unspecified behavior 43](#_Toc9605490)

[1.5 Undefined behavior 47](#_Toc9605491)

[1.6 Processing system-dependent behavior 66](#_Toc9605492)

[2. Functional Specifications 74](#_Toc9605493)

[2.1 Introduction 74](#_Toc9605494)

[2.1.1 About this Document 74](#_Toc9605495)

[2.2 About the Product 74](#_Toc9605496)

[2.2.1 Position of Product 74](#_Toc9605497)

[2.2.2 Feature 74](#_Toc9605498)

[2.2.3 Target Device 74](#_Toc9605499)

[2.2.4 Supported Environment 74](#_Toc9605500)

[2.2.5 System Configuration 75](#_Toc9605501)

[2.2.6 Package Configuration 75](#_Toc9605502)

[2.3 Outline of Processing 76](#_Toc9605503)

[2.3.1 Processing Flow 76](#_Toc9605504)

[2.4 Input/Output File 78](#_Toc9605505)

[2.4.1 Input File 78](#_Toc9605506)

[2.4.1.1 Types of input file 78](#_Toc9605507)

[2.4.1.2 C source file 78](#_Toc9605508)

[2.4.1.3 Assembly source file 78](#_Toc9605509)

[2.4.2 Output File 78](#_Toc9605510)

[2.4.2.1 Types of output file 78](#_Toc9605511)

[2.4.2.2 Object file 78](#_Toc9605512)

[2.4.2.3 Temporary file 78](#_Toc9605513)

[2.4.3 Command Line Format 79](#_Toc9605514)

[2.4.4 Options 79](#_Toc9605515)

[2.4.4.1 List of Options 79](#_Toc9605516)

[2.4.4.2 Detailed Descriptions of Options 80](#_Toc9605517)

[3. C99 Library 94](#_Toc9605518)

[3.1 Standard C99 Library 94](#_Toc9605519)

[3.1.1 Overview of C99 Library 94](#_Toc9605520)

[(1) Library Types 94](#_Toc9605521)

[(2) Organization of Library Part 94](#_Toc9605522)

[(3) Terms Used in Library Function Descriptions 95](#_Toc9605523)

[(4) Notes on Use of Libraries 96](#_Toc9605524)

[3.1.1.2 <assert.h> 97](#_Toc9605525)

[3.1.1.3 <errno.h> 98](#_Toc9605526)

[3.1.1.4 <fenv.h> 99](#_Toc9605527)

[3.1.1.5 <float.h> 101](#_Toc9605528)

[3.1.1.6 <limits.h> 103](#_Toc9605529)

[3.1.1.7 <math.h> 104](#_Toc9605530)

[3.1.1.8 <stdbool.h> 126](#_Toc9605531)

[3.1.1.9 <stddef.h> 127](#_Toc9605532)

[3.1.1.10 <stdint.h> 128](#_Toc9605533)

[3.1.1.11 <stdlib.h> 131](#_Toc9605534)

[3.1.1.12 mem<string.h> 133](#_Toc9605535)

[3.1.1.13 <iso646.h> 137](#_Toc9605536)

# Language Specification

## Basic language specification

### Conforming language specification

ccimp V3.00 is a compiler compliant with the following C language specification:

ISO/IEC 9899:1990 (JIS X 3010-2003, referred to as C99 in the remainder of this document)

### Constraints on the conforming language specification

Although ccimp V3.00 is a complier compliant with C99, it is subject to constraints on the language specification due to the instruction set architecture. The following describes the constraints.

In the following descriptions, note that all variables having static storage duration are called "global variables" - that is, global variables include variables with static storage class specifiers. All global variables described in this document are assumed to be "non-const-qualified global variables" unless otherwise described.

#### main function

This compiler does not handle the function named main as the program start processing function. For the main function, the user can specify desired parameters and return values. The main function can be called from another function.

The program start processing function must be declared and determined by the user through a pragma directive (#pragma entry) according to the extended language specification.

#### Constraints on types

The char type is handled as an 8-bit signed integer type.

For floating-point numbers, the following types can be written.

float, double and long double

However, this compiler handles the double and long double types as single-precision floating-point numbers.

A complex type is not allowed.

A variable-length array type is not allowed.

Bit-field members are not allowed in the structure type and union type.

#### Constraints on keywords

The following keywords are not allowed. Identifiers whose names are the same as the following keywords are also not allowed.

\_Complex, \_Imaginary, inline

#### Constraints on identifiers

The predefined identifier \_\_func\_\_ cannot be described. In addition, it is also impossible to describe the identifier named \_\_func\_\_.

#### Constraints on operations

The following integer arithmetic operations are not allowed. However, if both operands are constant and the divisor is not 0, it can be written.

%%=

If the divisor is not an integer constant or if the divisor is 0, the following integer arithmetic operations are not allowed.

//=

Long long type/ Type conversion between the unsigned long long type and the float type cannot be described.

Long long type/ Type conversion from the unsigned long long type to \_Bool type cannot be described.

Long long type/ Right or wrong of the description is shown in Table 1.1.2.5

Table 1.1.2.5. Operation of the unsigned long long/long long type.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # | Operator | | Operation specification right or wrong | Example of C source | Remarks |
| 1 | Postposition operator | id++ | Impropriety | long long ll,ll1;ll = ll1++; | It is enabled that it is a description because it becomes 64bit addition instruction. |
| 2 | id-- | Impropriety | long long ll,ll1;ll = ll1--; | It is enabled that it is a description because it becomes 64bit addition instruction. |
| 3 | [id] | Acceptable | long long ll,ll1;char ary[100];ll = ary[ll1]; | Operator with array affixing character. The specification of 64bit type is permitted as an array affixing character because it doesn't become 64bit integer type operation. |
| 4 | Function name (id) | Acceptable | long long ll,ll1;long long func(long long ll2);ll = func(ll1); | 64bit integer type specification of the function return type and the function actual argument is permitted because it becomes load/store instruction. |
| 5 | st.id; | Acceptable | long long ll,ll1;struct {long long sll;}st;ll = st.sll; | Structure/common body member reference operator. The description is permitted because it doesn't become 64bit integer type operation instruction. |
| 6 | stp->id; | Acceptable | long long ll,ll1;struct {long long sll;}\*stp;ll = stp->sll; | Structure/common body member reference operator. The description is permitted because it doesn't become 64bit integer type operation instruction. |
| 7 | (type name){ initialization [konara] } | Acceptable | long long \*ll = (long long[]){1,2,3}; | Compound literal. The description is permitted because it doesn't become 64bit integer type operation instruction. |
| 8 | Prepositive operator | ++id | Impropriety | long long ll,ll1;ll = ++ll1; | It is enabled that it is a description because it becomes 64bit addition instruction. |
| 9 | --id | Impropriety | long long ll,ll1;ll = --ll1; | It is enabled that it is a description because it becomes 64bit subtraction instruction. |
| 10 | Sizeof operator | sizeof | Acceptable | long long ll,ll1;ll = sizeof(ll1);ll1 = sizeof(long long); | The description is permitted because it doesn't become 64bit integer type operation instruction. |
| 11 | Unary operator | & | Acceptable | long long ll,\*llp;llp = &ll1; | Acquisition of address in 64bit type area. The description is permitted because it doesn't become 64bit integer type operation instruction. |
| 12 | \* | Acceptable | long long ll,\*llp;ll = \*llp; | Refer to the pointer in 64bit type area. The description is permitted because it doesn't become 64bit integer type operation instruction. |
| 13 | (type name) | Part is improper. | long long ll;unsigned long long ull;int \*ip;float f;\_\_Bool c;ll = (long long)ull;ll = (long long)ip;ll = (long long)f; // error c = Ll // error | Cast operator. Cast of the integer type and the pointer type permits describing because it doesn't become 64bit integer type operation instruction. However, warning is output about Cast with the pointer type. It is enabled that it is a description because there is no corresponding 64bit operation instruction. It is enabled that it is a description because it generates 64bit operation instruction (ne). |
| 14 | + | Acceptable | long long ll,ll1;ll = +ll1; | Monadic +. The description is permitted because it doesn't become 64bit integer type operation instruction. |
| 15 | - | Impropriety | long long ll,ll1;ll = -ll1; | It is enabled that it is a description because it becomes 64bit subtraction instruction. |
| 16 | ! | Impropriety | long long ll,ll1;ll = !ll1; | It is enabled that it is a description because of 64bit comparison instruction (ne). |
| 17 | ~ | Acceptable | long long ll,ll1;ll = ~ll1; | The description is permitted because it becomes INV instruction. |
| 18 | Clause 2 operator | + | Impropriety | long long ll,ll1,ll2;ll = ll1 + ll2; | It is enabled that it is a description because it becomes 64bit addition instruction. |
| 19 | - | Impropriety | long long ll,ll1,ll2;ll = ll1 - ll2; | It is enabled that it is a description because it becomes 64bit subtraction instruction. |
| 20 | \* | Impropriety | long long ll,ll1,ll2;ll = ll1 \* ll2; | It is enabled that it is a description because it becomes 64bit multiply instruction. |
| 21 | / | Impropriety | long long ll,ll1,ll2;ll = ll1 / ll2; | It is enabled that it is a description because it becomes 64bit division instruction. It is enabled that it is a description when the divisor is a constant. |
| 22 | % | Impropriety | long long ll,ll1,ll2;ll = ll1 % ll2; | It is enabled that it is a description because of the surplus calculation. |
| 23 | >> | Impropriety | long long ll,ll1,ll2;ll = ll1 >> ll2; | It is enabled that it is a description because it becomes 64bit shift instruction. |
| 24 | << | Impropriety | long long ll,ll1,ll2;ll = ll1 << ll2; | It is enabled that it is a description because it becomes 64bit shift instruction. |
| 25 | Relational operator | < | Impropriety | long long ll,ll1,ll2;ll = (ll1 < ll2); | It is enabled that it is a description because it becomes 64bit comparison instruction. |
| 26 | > | Impropriety | long long ll,ll1,ll2;ll = (ll1 > ll2); | It is enabled that it is a description because it becomes 64bit comparison instruction. |
| 27 | <= | Impropriety | long long ll,ll1,ll2;ll = (ll1 <= ll2); | It is enabled that it is a description because it becomes 64bit comparison instruction. |
| 28 | >= | Impropriety | long long ll,ll1,ll2;ll = (ll1 >= ll2); | It is enabled that it is a description because it becomes 64bit comparison instruction. |
| 29 | Equality operator | == | Impropriety | long long ll,ll1,ll2;ll = (ll1 == ll2); | It is enabled that it is a description because it becomes 64bit comparison instruction. |
| 30 | != | Impropriety | long long ll,ll1,ll2;ll = (ll1 != ll2); | It is enabled that it is a description because it becomes 64bit comparison instruction. |
| 31 | Bit operator | & | Acceptable | long long ll,ll1,ll2;ll = (ll1 & ll2); | The description is permitted for the settlement for 64bit AND instruction. |
| 32 | ^ | Acceptable | long long ll,ll1,ll2;ll = (ll1 ^ ll2); | The description is permitted for the settlement for 64bit XOR instruction. |
| 33 | | | Acceptable | long long ll,ll1,ll2;ll = (ll1 | ll2); | The description is permitted for the settlement for 64bit OR instruction. |
| 34 | Boolean | && | Impropriety | long long ll,ll1,ll2;ll = (ll1 && ll2); | It is enabled that it is a description because it generates 64bit comparison instruction (ne) with both operands of the boolean. Moreover, when both operands are 64bit integer types, the same error is output by both operands. |
| 35 | || | Impropriety | long long ll,ll1,ll2;ll = (ll1 || ll2); | It is enabled that it is a description because it generates 64bit comparison instruction (ne) with both operands of the boolean. Moreover, when both operands are 64bit integer types, the same error is output by both operands. |
| 36 | Conditional operator | ? : | Part is improper. | long long ll,ll1,ll2;int i;float f;ll = (ll ? ll1 : ll2);i ? f : ll;// Cast to the float type is generated in ll. | The description in the first operand is assumed to be improper because it becomes 64bit operation instruction (ne). |
| 37 | Simple assignment operator | = | Acceptable | long long ll,ll1;ll = ll1; | The description is permitted because it becomes load/store. |
| 38 | Compound assignment operator | += | Impropriety | long long ll,ll1;ll += ll1; | It is enabled that it is a description because it becomes 64bit addition instruction. |
| 39 | -= | Impropriety | long long ll,ll1;ll -= ll1; | It is enabled that it is a description because it becomes 64bit subtraction instruction. |
| 40 | \*= | Impropriety | long long ll,ll1;ll \*= ll1; | It is enabled that it is a description because it becomes 64bit multiply instruction. |
| 41 | /= | Impropriety | long long ll,ll1;ll /= ll1; | It is enabled that it is a description because it becomes 64bit division instruction. It is enabled that it is a description when the divisor is a constant. |
| 42 | %= | Impropriety | long long ll,ll1;ll %= ll1; | It is enabled that it is a description because it becomes a surplus calculation. |
| 43 | <<= | Impropriety | long long ll,ll1;ll <<= ll1; | It is enabled that it is a description because it becomes 64bit shift instruction. |
| 44 | >>= | Impropriety | long long ll,ll1;ll >>= ll1; | It is enabled that it is a description because it becomes 64bit shift instruction. |
| 45 | &= | Acceptable | long long ll,ll1;ll &= ll1; | The description is permitted for the settlement for 64bit AND instruction. |
| 46 | ^= | Acceptable | long long ll,ll1;ll ^= ll1; | The description is permitted for the settlement for 64bit XOR instruction. |
| 47 | |= | Acceptable | long long ll,ll1;ll |= ll1; | The description is permitted for the settlement for 64bit OR instruction. |

#### Constraints on pointer types

A pointer type pointing to a function is not allowed.

#### Constraints on constants

This compiler handles floating-point constants without a suffix or with suffix l, L or F as single-precision floating-point numbers.

Wide character constants are not allowed.

#### Constraints on string literals

String literals are not allowed.

#### Constraints on compound literals

Compound literals are not allowed for initializing global variables.

#### Constraints on address constants

The address constants for function names are not allowed.

This means that function names cannot be assigned to pointer type variables.

#### Constraints on data and function declarations

Initial values cannot be described for global variables that do not have const qualification.

A const-qualified global variable cannot be described other than 4 byte long scalar type and 4 byte long scalar type array.

Using ellipsis (…) for data and functions declarations are not allowed. This means that any functions that receive a variable number of arguments are not allowed.

The inline function specifier is not allowed.

The old format (K&R format) is not allowed.

#### Constraints on preprocessor directives

In the identifier list of a macro definition, using ellipsis (…) is not allowed.

#### Notes on the initial value of a global variable

For a const-qualified global variable, the initial value written in C programs may not match the value loaded during program execution. Values may have already been set in uninitialized global variables before program execution.

This is because the image recognition system executes preprocessing in which initial values can be specified in desired global variables before execution of the Shader program written in a C source file.

The user must create Shader programs while taking into consideration which global variables are set to initial values in the preprocessing before execution of the Shader programs.

#### Instruction output specification for a right shift operation

A right shift operation on the unsigned integer type results in a logical right shift instruction.

A right shift operation on the signed integer type results in an arithmetic right shift instruction.

#### Restrictions on library functions

When using the math library, it is necessary to call initmath functions in advance to perform initialization.

When using the rand function, it is necessary to call the srand function in advance to set the seed of the pseudo random number sequence.

### Floating-point operations

Floating-point operations in ccimp conform to ANSI/IEEE Std 754-1985 (hereafter called IEEE754) for the most part, but the following items are not complied with.

#### Constraints on rounding mode

IEEE754 prescribes five rounding modes but the compiler does not support all modes.

For the rounding mode used for optimization in the compiler, only the round-to-nearest (to the nearest even) or round-to-zero mode can be selected.

### Restrictions on formulas

Each control statement cannot have long long type / unsigned long long type operands.

Therefore, long long type / unsigned long long type operands cannot be described in the control expression operands shown in Table 1.1.4.

Table 1.1.4 long long type / unsigned long long type operand not described control expression

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # | Expression statement | | Right or wrong of description | Example of C source | Remarks |
| 1 | Controlling expression | if statement controlling expression | Impropriety | long long ll;if (ll) { ……} | It is enabled that it is a description because it generates 64bit operation instruction (ne). |
| 2 | for statement controlling expression | Impropriety | long long ll;for (; ll; ) { ……} | It is enabled that it is a description because it generates 64bit operation instruction (ne). |
| 3 | while statement controlling expression | Impropriety | long long ll;while (ll) { ……} | It is enabled that it is a description because it generates 64bit operation instruction (ne). |
| 4 | do-while statement controlling expression | Impropriety | long long ll;do { ……}while (ll); | It is enabled that it is a description because it generates 64bit operation instruction (ne). |
| 5 | switch statement controlling expression | Impropriety | long long ll;switch (ll) { ……} | It is enabled that it is a description because it generates 64bit comparison operation instruction. |

## Runtime environment

### Internal representation of data

#### Integer types

Table 1.1 gives the data types that can be used and their valid value ranges.

Table 1.1 Integer types that can be used and their valid value ranges

|  |  |
| --- | --- |
| Type name | Valid value range |
| char  signed char | -128 to 127 |
| unsigned char | 0 to 255 |
| short  signed short | -32768 to 32767 |
| unsigned short | 0 to 65535 |
| int  signed int  long  signed long | -2147483648 to 2147483647 |
| unsigned int  unsigned long | 0 to 4294967295 |
| long long  signed long long | -9223372036854775808 to 9223372036854775807 |
| unsigned long long | 0 to 18446744073709551615 |

#### Floating-point type

The internal representation of the floating-point type conforms to the IEEE754 format.

Table 1.2 gives the data types that can be used and their valid value range, and Figure 1.1 shows the internal representation of the data type.

Table 1.2 Floating-point types that can be used and their valid value range

|  |  |
| --- | --- |
| Type name | Valid value range |
| float  double  long double | 1.17549435E-38F to 3.40282347E+38F  \* Excluding denormal numbers. |

Sign bit (1 bit)

Exponent part (8 bits) Mantissa (23 bits)

31 30 23 22 0

Figure 1.1 Internal representation of float

#### Pointer types

The internal representation of the pointer types is a 32-bit unsigned integer.

#### Enumerated types

The internal representation of the enumerated types is the same as that of int.

#### Array types

The internal representation of the array types is an arrangement of array elements in such a way to meet the alignment conditions for the elements.

#### Structure types

The internal representation of the structure types is an arrangement of structure members in such a way to meet the alignment conditions for the members.

#### Union types

The internal representation of the union types is one in which the maximum member size is set as the size of the union and the base positions of all members are set to an offset of 0.

#### Bit fields

In this compiler, bit fields cannot be used.

#### Alignment conditions

For the 1-byte scalar type, the alignment value is 1 (placed at 1-byte boundaries).

For the 2-byte scalar type, the alignment value is 2 (placed at 2-byte boundaries).

For the 4-byte scalar type, the alignment value is 4 (placed at 4-byte boundaries).

For the 8-byte scalar type, the alignment value is 8 (placed at 8-byte boundaries).

For the vector type, which is included in the extended language specification, refer to section 1.3.1, Vector types.

The alignment value for the array type is the same as that for the type of its elements.

For the structure type and union type, the maximum alignment value for the data types of their members is used.

### Sections

#### List of sections

Table 1.3 lists the sections to be generated by the compiler.

Table 1.3 Sections

|  |  |
| --- | --- |
| Section name | Description |
| .text | Program area.  Instructions of the functions written in the C program are placed in this section. |
| impc\_comm | Rewritable data area. This corresponds to Scratchpad (common) in the Shader memory map.  Uninitialized global variables can be placed in this section through the pragma directive #pragma section in the order in which they are defined. |
| impc | Rewritable data area. This corresponds to Scratchpad (non-common) in the Shader memory map.  Uninitialized global variables can be placed in this section through the pragma directive #pragma section in the order in which they are defined.  This section may also be used for the stack to handle temporary variables or arguments to be passed via memory. |
| lwm\_comm | Rewritable data area. This corresponds to LWM (common) in the Shader memory map.  Uninitialized global variables are placed in this section by default in the order in which they are defined. |
| gwm\_comm | Rewritable data area. This corresponds to GWM (common) in the Shader memory map.  Uninitialized global variables can be placed in this section through the pragma directive #pragma section in the order in which they are defined. |
| uniform | Data area that cannot be rewritten from a C program. This corresponds to the constant memory area in the Shader memory map.  const-qualified global variables are placed in this section in the order in which they are defined.  When the constant area has unused space after all const-qualified global variables are placed, the compiler also places in this section the constant values written in C programs. Constant values are placed after the last global variable definition.  Constant values are placed during assembly, so they do not appear in this section in assembly source code. |
| lwm | Rewritable data area. This corresponds to LWM (non-common) in the Shader memory map.  Uninitialized global variables can be placed in this section through the pragma directive #pragma section in the order in which they are defined.  This section may also be used for the stack to handle temporary variables or arguments to be passed via memory. |
| gwm | Rewritable data area. This corresponds to GWM (non-common) in the Shader memory map.  Uninitialized global variables can be placed in this section through the pragma directive #pragma section in the order in which they are defined. |

#### Notes on memory access to lwm and lwm\_comm

The instruction set of IMP-X5+ has 8-byte load instructions (LDLL, LDLLI, LDLRB and LDLRBI) and store instructions (STLL, STLLI, STLRB and STLRBI). These instructions can be used when the access destination is a section that corresponds to LWM in the Shader memory map.

If the section that a global variable or stack belongs to is lwm or lwm\_comm, the compiler assumes that the access destination memory is LWM, and that an 8-byte load or store instruction can be generated. Note that the compiler determines the memory from the section at the beginning of a variable. Therefore, if source code that accesses another section by using the position of the variable written in LWM as the base position, the compiler generates an invalid 8-byte load or store instruction.

Example of source code (1) shows an example of source code that generates an invalid 8-byte load or store instruction. In this example, the compiler generates an STLLI instruction for a write to memory by using an array that begins with gv. In this case, if the user who coded the program has assumed that the write destination is outside LWM, the program does not run as intended because the STLLI instruction cannot access memory correctly.

Example of source code (1): source code that generates an invalid 8-byte load or store instruction

\_\_int8x8 gv; // gv is in the lwm section

void f() {

(&gv)[10000] = gv; // STLLI is generated even if the write destination is not LWM.

}

### Function call interface

#### Using the Processor registers

Table 1.4 lists the Processor registers and their uses. General-purpose registers can be classified into two types; the registers whose values are not guaranteed before and after a function call (caller-save), and the registers whose values are guaranteed before and after a function call (callee-save).

The compiler uses Processor register R7 for the operand of the call instruction. As the operand of the call instruction is placed in R7, the operand of the return instruction (brar) in the function to be called by the call instruction is also placed in R7.

Table 1.4 Processor registers

|  |  |
| --- | --- |
| Register | Use |
| R0 | X-coordinate / general-purpose register (callee-save) |
| R1 | Y-coordinate / general-purpose register (callee-save) |
| R2 | Stack pointer |
| R3 | Zero register |
| R4 to R5 | General-purpose registers (caller-save) |
| R6 | Uniform index register / general-purpose register (caller-save) |
| R7 | General-purpose registers (callee-save)  and the register for the operand of the call and brar instruction |
| R8 to R15 | General-purpose registers (caller-save) / SOU (Shared Operation Unit) |
| R16 to R25 | General-purpose registers (caller-save) |
| R26 to R31 | General-purpose registers (callee-save) |
| R32 to R39 (V3U) | General-purpose registers (caller-save) |
| R40 to R47 (V3U) | General-purpose registers (callee-save) |

#### Rules for setting for arguments and using parameters

To pass an argument via a register, R16, R17, R18, R19, R20, R21, R22, R23, R24 or R25 is used.

A 4-byte scalar-type or 4-byte vector-type argument is passed through one register.

An 8-byte scalar-type or 8-byte vector-type argument is passed via a register pair. Details of a vector-type argument are described in section 1.3.1, Vector types.

Even for a type that can be passed via a register, the argument is passed via a stack when no register is available.

A structure-type or union-type argument is always passed via a stack.

When an argument is passed via a stack, the value of the argument is written to the stack according to the size and alignment value of the argument type.

Whether an argument (argument a) can be passed via a register is determined as follows.

* When argument a is a structure type or a union type, it is passed via a stack.
* When argument a is the first argument and is a type that can be passed via a register, it is passed via the smallest-number register.
* When any argument written to the left of argument a is passed via register R25, argument a is passed via a stack.
* When the argument written to the left of argument a is passed via a stack, argument a is passed via a stack.
* In all other cases, argument a is passed via the register with the next number after the register used for the left-side argument (if the register with the next number cannot be used, argument a is passed via a stack).

[Examples] \* ST is a structure type and \_\_int8x8 is an 8-byte vector type.

void f1(int a, int b, int c, int d);

→ a is passed via register R16, b is passed via register R17 (next number of the register for the left-side argument), c is passed via register R18, and d is passed via register R19.

void f2(\_\_int8x8 a, \_\_int8x8 b, \_\_int8x8 c, \_\_int8x8 d, \_\_int8x8 e, int f);

→ a, b, c, d, e are passed via registers R16 and R17, R18 and R19, R20 and R21, R22 and R23, R24 and R25 correlatively.

f is passed via a stack because R25 is used for the left-side argument.

void f3(ST a, int b, int c, int d);

→ a is passed via a stack because it is a structure type.

b, c, and d are passed via a stack because the left-side argument is passed via a stack.

void f4(int a, \_\_int8x8 b, \_\_int8x8 c, \_\_int8x8 d, \_\_int8x8 e, int f);

→ a, b, c, d, e are passed via registers R16, R18 and R19, R20 and R21, R22 and R23, R24 and R25 correlatively.

f is passed via a stack because R25 is used for the left-side argument.

void f5(int8x8 a, \_\_int8x8 b, \_\_int8x8 c, \_\_int8x8 d, int e, int f);

→ a, b, c, d, e are passed via registers R16 and R17, R18 and R19, R20 and R21, R22 and R23, R24 correlatively.

f is passed via a stack because no register is available to pass an 8-byte vector-type argument.

#### Rules for setting and using a return value

For a return value of a scalar type or a 4-byte vector type, the value is returned to the caller via a register. Register R16 is used in this case.

For a return value of an 8-byte vector type, the value is returned to the caller via a pair registers. Registers R16 and R17 are used in this case.

For a return value of a structure type or a union type, the value is returned to the caller by writing the value to the location indicated by the structure-type return value pointer. The caller of a function must reserve an area where the return value is to be written, and specify the address of this area in the structure-type return value pointer. The callee must write the return value to the address indicated by the structure-type return value pointer.

R4 is used as the structure-type return value pointer.

#### Rules for built-in function \_\_getX and \_\_getY

In the IMP-X5+ register specifications, register R0 has the meaning of the X coordinate, and register R1 has the meaning of the Y coordinate. In C source code, built-in functions \_\_getX and \_\_getY are used to read the contents of those registers, and built-in functions \_\_setX and \_\_setY are used to write data to those registers. In addition, several instructions generated by these built-in functions implicitly reference R0 and R1. The details of built-in functions are described in section 1.3.3, Built-in functions.

No instructions generated by the compiler implicitly rewrite the contents of registers R0 and R1. The following describes the contents by using a C source code as an example.

Example of source code (2): source code that obtains the contents of R0

int x;

void func2();

void func() {

x = \_\_getX(); //(a)

func2();

x = \_\_getX(); //(b)

}

The code generated by the compiler guarantees the following:

(1) If there is no source code that rewrites R0 (that is, built-in function \_\_setX is not used) within the processing scope of function func2(), the value obtained by (a) and the value obtained by (b) are the same.

(2) If there is source code that rewrites R0 (that is, built-in function \_\_setX is used) within the processing scope of function func2(), the value obtained by (b) is the same as the value that is last set by \_\_setX within function func2().

If the information about the X and Y coordinates is not required (that is, no read and write operations for R0 and R1 are performed) in a C source code, the compiler can use R0 and R1 as general-purpose registers. Therefore, the compiler can use R0 and R1 as general-purpose registers as part of optimization.

The compiler performs optimization that enables use of R0 and R1 as general-purpose registers only when these registers can be used as general-purpose registers without any problems. User programs coded guaranteeing (1) and (2) above do not cause any side effects as a result of optimization. No users need to worry about impacts of instructions generated by the compiler when writing built-in functions related to R0 and R1 in source code.

#### Interface between functions for R8 to R15

In the IMP-X5+ register specifications, the roles of Processor registers R8 to R15 can be changed by writing data to control registers (CR19 and CR20). R8 to R15 can be used as general-purpose registers or SOUs (Shared Operation Units).

The compiler uses R8 to R15 as general-purpose registers (caller-save registers), and generates code so that the following inter-function interface conditions are met:

* At the beginning of the program start processing function (entry function), whether R8 to R15 are general-registers or SOUs is undefined.
* When processing reaches the beginning of a normal function (function that is not the entry function), all of R8 to R15 are general-purpose registers.
* After function A calls function B, when processing has just returned from function B to function A, all of R8 to R15 are general-purpose registers.

Assembly source programs called by, or that call, compiler-generated programs must also be coded so that the above inter-function interface conditions are met. The above interface conditions must be met when an assembly source program is written in an assembly source file, and when an assembly source program is written in a C source file by using the extended language specification.

#### Rules for stacks

The compiler uses Processor register R2 as a stack pointer.

The compiler outputs an instruction that initializes the stack pointer at the beginning of the program start processing. When initialized, the stack pointer is set to the bottom of the stack used (maximum address + 1).

To use a stack, decrement the stack pointer value by the size to use, and reserve an area.

After finishing the use of the stack, increment the stack pointer value by the size that is no longer necessary, and release the area.

The compiler generates code, assuming that the beginning of the stack frame of each function is aligned by 8 bytes. The compiler controls the stack pointer so that the stack frame size for each function is always a multiple of 8.

Figure 1.2 shows a stack layout immediately after a function call if one function (caller) calls another function (callee).

Fifth argument

Sixth argument

…

n-th argument

Area allocated for auto variables

Stack frame of

the caller function

Stack used by

the callee function

Toward address 0 ↑

← SP position

Higher addresses ↓

Area for saving callee-save registers

Figure 1.2 Stack layout immediately after a function call

#### External names

The compiler handles as external names the function names and global variable names that are not declared as static. In the assembly source file, external names will become label names with "\_" added to the beginning of the names.

## Extended language specification

### Vector types

#### Vector types that can be used

Table 1.5 shows the vector types that are allowed in ccimp V3.00.

Table 1.5 Vector types

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Vector type name | Element type | Vector type size | Number of elements | Aligned by (bytes) |
| \_\_int8x8 | signed char | 8 byte | 8 | 8 |
| \_\_int8x4 | signed char | 4 byte | 4 | 4 |
| \_\_uint8x8 | unsigned char | 8 byte | 8 | 8 |
| \_\_uint8x4 | unsigned char | 4 byte | 4 | 4 |
| \_\_int16x4 | signed short | 8 byte | 4 | 8 |
| \_\_int16x2 | signed short | 4 byte | 2 | 4 |
| \_\_uint16x4 | unsigned short | 8 byte | 4 | 8 |
| \_\_uint16x2 | unsigned short | 4 byte | 2 | 4 |
| \_\_int32x2 | signed int | 8 byte | 2 | 8 |
| \_\_uint32x2 | unsigned int | 8 byte | 2 | 8 |

#### Loading or storing vectors

For the order of vector elements in the registers, vector elements with lower addresses and higher addresses in memory are listed on the LSB side and the MSB side, respectively.

For 8-byte vectors, an element group of addresses for the lower-order 4 bytes and an element group for the higher-order 4 bytes in memory are assigned to even- and odd-numbered registers in register pairs, respectively.

The compiler loads and stores vectors so that the above ordering of vector elements is guaranteed.

The address indicated by the pointer to a vector type must be boundary-aligned according to the alignment for the vector type. Guaranteeing this is the responsibility of the user.

Example of source code (3) and Figure 1.3 and Figure 1.4 show examples of the order of data for an 8-byte vector.

Example of source code (4) and Figure 1.5 and Figure 1.6 show examples of the order of data for a 4-byte vector.

Example of source code (5) shows an example of a case where boundaries must be aligned in the source code.

Example of source code (3): source code for an 8-byte vector argument

\_\_int8x8 gv;

\_\_int8x8 func1(void) {

\_\_int8x8 x = {0, 1, 2, 3, 4, 5, 6, 7};

gv = x;

return x;

}

0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07

…

→

Higher addresses

…

←

Lower addresses

Figure 1.3 Order of data of array gv in memory in example of source code (3)

0x03 0x02 0x01 0x00

0x07 0x06 0x05 0x04

r17

r16

↑ LSB

↑ MSB

Figure 1.4 Order of data in return-value registers r16 and r17 for function func1 in example of source code (3)

Example of source code (4): source code for a 4-byte vector argument

\_\_int16x2 gv;

\_\_int16x2 func2(void) {

\_\_int16x2 x = {0x0123, 0x4567};

gv = x;

return x;

}

0x23 0x01 0x67 0x45

…

→

Higher addresses

…

←

Lower addresses

Figure 1.5 Order of data of array gy in memory in example of source code (4)

0x45 0x67 0x01 0x23

r16

↑ LSB

↑ MSB

Figure 1.6 Order of data in return-value register r16 for function func2 in example of source code (4)

Example of source code (5): source code in a case where boundaries must be guaranteed

typedef union {

signed char data[8];

\_\_int8x8 dummy;

} DATA8;

DATA8 gv;

\_\_int8x8\* func3() {

return (\_\_int8x8\*)(gv.data);

}

The array is 1-byte aligned. If the beginning of the array is positioned at an 8-byte boundary by using dummy union members, the beginning address of this array can be used for a pointer to a vector type.

#### Reading and writing vector elements

Table 1.6 shows a list of the subscripts allowed for vector-type variables. Using subscripts enables reading or writing of the elements of vectors. Subscripts not shown in Table 1.6 are not usable.

The first element is allocated to the LSB side of the vector data in the register. For 8-byte vectors, the first element is the LSB of the odd-numbered register in a register pair.

Subscript s may be either lower-case or upper-case.

Table 1.6 Subscripts allowed for vector-type variables

|  |  |
| --- | --- |
| Subscript | Description |
| .s0 | Indicates the first element of a vector-type variable. |
| .s1 | Indicates the second element of a vector-type variable. |
| .s2 | Indicates the third element of a vector-type variable. |
| .s3 | Indicates the fourth element of a vector-type variable. |
| .s4 | Indicates the fifth element of a vector-type variable. |
| .s5 | Indicates the sixth element of a vector-type variable. |
| .s6 | Indicates the seventh element of a vector-type variable. |
| .s7 | Indicates the eighth element of a vector-type variable. |

0x03 0x02 0x01 0x00

0x07 0x06 0x05 0x04

r17

r16

↑ .s7 ↑ .s6 ↑ .s5 ↑ .s4

↑ .s3 ↑ .s2 ↑ .s1 ↑ .s0

Figure 1.7 Positions of elements in registers r16 and r17 when they hold \_\_int8x8-type data

0x0302 0x0100

r16

↑ .s1 ↑ .s0

Figure 1.8 Positions of elements in register r16 when it holds \_\_int16x2-type data

#### Constraints on vectors

The following lists cases where vectors are allowed in source code. If none of cases (1) to (9) below are applicable, the use of vectors is not allowed.

(1) Vector-type variables having static storage duration or automatic storage duration are allowed.

(2) Vector-type automatic variables can be initialized during declaration by using a vector-type variable or an initializer for which constants are listed in curly brackets ({}).

Example of source code (6): initialization of vector-type variables

void func2(\_\_int8x8, \_\_int8x8, \_\_int8x8);

void func(\_\_int8x8 a) {

\_\_int8x8 tmp1 = a;

\_\_int8x8 tmp2 = {0,1,2,3,4,5,6,7}; // Initial values of elements s0, s1, s2, s3,

// s4, s5, s6, and s7 are 0, 1, 2, 3, 4, 5,

// 6, and 7, respectively.

\_\_int8x8 tmp3 = {0,1}; // If Number of listed constants < Number of vector-type

// elements, elements whose initial values are unknown are

// initialized to 0.

func2(tmp1, tmp2, tmp3);

}

(3) Arrays of vectors are allowed.

(4) For the structure type and union type, vector-type members can be written.

(5) Addresses of vectors may be acquired.

(6) Pointers to vectors are allowed.

(7) Vectors are allowed as parameters of functions (passing as parameters is allowed).

(8) Vectors are allowed as return values of functions (passing as return values is allowed).

(9) Vector-type variables are allowed as operands of the following operators.

- Simple assignment by the assignment operator: =

- Unary address operator: &

- The second and the third operands of the conditional operator: ?

- sizeof operator

- Comma operator

(10) The same size vector type, integer type, and type conversion between vector types can be described implicitly or explicitly.

However, even if cases of (1) to (9) above in which vectors are allowed apply, the following restrictions disallow vectors in source code.

* Type qualifiers (const and volatile) are not allowed for vectors.
* Vector types cannot be type converted to data types other than integer types, either implicit or explicit.
* Vector types cannot be converted to different sized data types, either implicitly or explicitly.
* When initializing a vector-type automatic variable by using a {} specification, no variables can be written in {}.
* When initializing a vector-type automatic variable by using a {} specification, no more constants than the number of vector-type elements can be written.
* For a structure or union that has vector-type members, initialization by member cannot be performed.
* Vector-type arrays cannot be initialized.

#### Restrictions on writing vector elements

The addresses of vector elements cannot be obtained.

Example of source code (7): obtaining the address of a vector-type element (code resulting in an error)

\_\_int8x8 v;

signed char\* func() {

return &(v.s0); // Obtaining an element address results in a compilation error.

}

### #pragma

#### #pragma inline, #pragma noinline

* Format

#pragma inline [(]<function name>[,…][)]

#pragma noinline [(]<function name>[,…][)]

* Description

#pragma inline is used to declare the function subject to inline expansion. The function declared with #pragma inline is regarded as being subject to inline expansion, regardless of any compile option.

#pragma noinline is used to declare the function not subject to inline expansion. The function declared with #pragma noinline is regarded as being subject to a function call, without being subject to inline expansion, regardless of any compile option.

* Example

C program:

#pragma inline func\_add

#pragma noinline func\_sub

int gv;

int func\_add(int a, int b) {

return a+b;

}

int func\_sub(int a, int b) {

return a-b;

}

void func(int a, int b, int c, int d) {

// func\_add is subjected to inline expansion, without being subjected

// to a function call.

gv = func\_add(a,b);

// func\_sub is subjected to a function call, without being subjected to

// inline expansion.

gv += func\_sub(c,d);

}

* Remarks

#pragma inline and #pragma noinline must be written before the function itself is defined.

#pragma inline does not guarantee that the declared function is always subject to inline expansion. The compiler may judge not to subject it to inline expansion, considering increases in compilation time and the amount of memory used.

#### #pragma inline\_asm

* Format

#pragma inline\_asm [(]<function-name>[(clobbers={<register-name-list>|none})][,...][)]

<register-name-list>: {<register-name>[,<register-name-list>]|

<register-name>-<register-name>[,<register-name-list>]|<special-register-name>}

<register-name>: r4 to r31 (r is case-insensitive.) for -cpu=v3m2 or v3h

r4 to r47 (r is case-insensitive.) for -cpu=v3u

<special-register-name>:LPC (LPC is case-insensitive.) for -cpu=v3u

* Description

The assembly code in the function declared with #pragma inline\_asm (referred to as the inline\_asm function in the remainder of this document) is subject to inline expansion. The function call rules for the inline\_asm function are the same as those for other functions written in C programs. By writing assembly code in the inline\_asm function in accordance with the function call rules, it is possible to receive arguments and set return values.

The register to be rewritten within function inline\_asm can be specified by using clobbers. When the compiler performs inline expansion for function inline\_asm during compilation, in accordance with the clobbers specification, the compiler assumes that the caller-save register can be used across the inline-expanded sections. Also, based on the clobbers specification, the compiler omits code generation for changing the roles (whether the register is a general-purpose register or SOU) of registers r8 to r15 in the inline-expanded sections.

If you specify none for clobbers, the compiler does not rewrite all general purpose registers and LPCs in the inline\_asm function.

If clobbers is not written, the compiler performs compilation, assuming that all caller-save registers are written in clobbers.

* Example (when both #pragma inline\_asm and clobbers are written)

C program:

// (1) If clobbers is not written, compiler assumes that all caller-save

// registers are subjected to rewriting during compilation.

#pragma inline\_asm f1

// (2) Compiler assumes that no general-purpose registers are subjected to

// rewriting during compilation.

#pragma inline\_asm f2(clobbers=none)

// (3) Compiler assumes that r16, r17, r18, and r19 caller-save registers are

// subjected to rewriting during compilation.

#pragma inline\_asm f3(clobbers=r16,r17,r18,r19)

// (4) Compiler assumes that r16, r17, r18, and r19 caller-save registers are

// subjected to rewriting during compilation. (Same as (3) above.)

#pragma inline\_asm f4(clobbers=r16,r17-r19)

// (5) Compiler assumes that all of r4 to r31 caller-save registers are subjected

// to rewriting during compilation.

#pragma inline\_asm f5(clobbers=r4-r31)

* Example (code generated by #pragma inline\_asm)

C program:

#pragma inline\_asm func\_add

int gv;

static int func\_add(int a, int b, int c, int d) {

add r16, r16, r17

add r16, r16, r18

add r16, r16, r19

}

void func() {

gv = func\_add(1,2,3,4);

}

Assembly language source code example as a result of compilation:

\_func:

movi r19, 0x00000004

movi r18, 0x00000003

movi r17, 0x00000002

movi r16, 0x00000001

\_line\_top inline\_asm

add r16, r16, r17

add r16, r16, r18

add r16, r16, r19

\_line\_end inline\_asm

movui r17, \_gv

sti r16, r17, 0x00000000

brar r7

Assembly language source code that has undergone inline expansion

* Example (when function inline\_asm does not rewrite caller saved registers)

C program:

#pragma inline\_asm func\_nop(clobbers=none)

Static void func\_nop() {

nop

}

void func2(int, int, int, int);

void func(int a, int b, int c, int d) {

func\_nop();

func2(a, b, c, d);

}

Assembly language source code example as a result of compilation:

\_func:

addi r2, r2, 0xFFFFFFF8

sti r7, r2, 0x00000004

\_line\_top inline\_asm

nop

\_line\_end inline\_asm

call r37, \_func2

ldi r7, r2, 0x00000004

addi r2, r2, 0x00000008

brar r7

The register that received parameters in function func and the register that is used to pass arguments in function func2 are the same, and clobbers is not used in function inline\_asm. Therefore, instructions for setting arguments in function func2 are omitted.

* Example (function inline\_asm rewrites only some caller saved registers)

C program:

#pragma inline\_asm func\_add(clobbers=r7-r16,r18)

Static int func\_add(int a, int b) {

mov r7, r16

add r18, r17, r7

mov r16, r18

}

void func2(int, int, int, int);

void func(int a, int b, int c, int d) {

a = func\_add(a, b);

func2(a, b, c, d);

}

Assembly language source code example as a result of compilation:

\_func:

addi r2, r2, 0xFFFFFFF8

sti r31, r2, 0x00000004

mov r20, r18

\_line\_top inline\_asm

mov r7, r16

add r18, r17, r7

mov r16, r18

\_line\_end inline\_asm

mov r18, r20

call r7, \_func2

ldi r7, r2, 0x00000004

addi r2, r2, 0x00000008

brar r7

Because r7, r16, and r18 are specified as clobbers in function inline\_asm, in the code for setting arguments in function func2, the code for setting arguments in r17 and r19 is omitted.

* Example (when an SOU is used in function inline\_asm)

C program:

#pragma inline\_asm func1

Static float func1(float a) {

stci cr20, 0x0000ff00 // Sets the register as an SOU.

...

stci cr19, 0x0000ff00 // Sets the register as a general-

// purpose register.

}

#pragma inline\_asm func2(clobbers=r13,r16)

Static float func2(float a) {

stci cr20, 0x00002000 // Sets the register as an SOU.

...

stci cr19, 0x00002000 // Sets the register as a general-

// purpose register.

}

If an SOU is used in function inline\_asm in which no clobbers is written, the cr19 and cr20 settings are written for all of r8 to r15.

If an SOU is used in function inline\_asm in which clobbers is written, the cr19 and cr20 settings need to be written for only the registers that are included in clobbers of registers r8 to r15 (in the example of func2, only r13 applies).

* Remarks

#pragma inline\_asm must be written before the function itself is defined.

The compiler outputs the assembly source code description in the inline\_asm function to assembly language source code without checking.

When writing labels in assembly language source code, make sure that there will be no duplicate label names in assembly language source code after inline expansion.

In the inline\_asm function, comments in the assembly language style (comments that start with a #) are not allowed.

To reference a function or a variable in the inline\_asm function, its external name with "\_" added at the beginning must be used.

If the inline\_asm function contains a CALLA instruction, correct program execution is not guaranteed.

If the inline\_asm function contains a CALL or CALLR instruction that calls a function declared as static, correct program execution is not guaranteed.

If the inline\_asm function contains an instruction that is not defined in the C source file, correct program execution is not guaranteed.

If callee-save registers need to be rewritten in function inline\_asm, the user must write register save and restore operations in the assembly source code of function inline\_asm.

If a function call instruction needs to be written in function inline\_asm, because the function call may rewrite all caller-save registers, clobbers must not be written. If written, operation of the program is not guaranteed.

To use registers r8 to r15 as SOUs if clobbers is written, the cr19 and cr20 settings can be changed for only the registers that are written in clobbers in the assembly language source code of the inline\_asm function definition. If the cr19 and cr20 settings are changed for registers that are not written in clobbers, operation of the program is not guaranteed.

#### #pragma stacksize

* Format

#pragma stacksize <stack size>

* Description

#pragma stacksize is used to declare a size in the stack area for use during program execution. The stack size must be specified in bytes.

If this #pragma directive is not specified in a program, the compiler sets aside a 128-byte area for use in the stack.

* Example

// To use a 256-byte area as the stack:

#pragma stacksize 0x100

* Remarks

The stack size value must be a multiple of 8 within the range of 0 to 32760.

The number described in the stack size must be a multiple of eight.

This #pragma directive can only be used once in a C source file; it must not be written more than once.

#### #pragma entry

* Format

#pragma entry <function name> <master|slave>

* Description

The function specified as <function name> is compiled as the program start processing function (entry function).

"master" is the option to indicate that the function is the start position of the Shader program execution that is activated by the host CPU.

"slave" is the option to indicate that the function is the start position of the Shader program execution that is activated by the ACTST instruction.

No return value can be specified in the entry function.

No parameter can be written in the master entry function.

In the slave entry function, scalar 4-byte parameter is allowed. When a parameter is written, its value is received via r2; this differs from the function call rules.

The compiler generates entry and exit instructions for the entry function as follows.

* Even when a callee-save register is used in the entry function, its value is not saved or restored.
* The instruction for initializing the stack pointer is output at the entry to the entry function.
* The trap instruction is output as an instruction corresponding to the return instruction at the exit from the entry function.
* Remarks

This pragma directive can be specified multiple times, that is, there can be multiple entry functions.

#### #pragma section

* Format

#pragma section bss <section>

<section>: lwm | lwm\_comm | gwm | gwm\_comm | impc | impc\_comm

* Description

The variables defined after this pragma directive are placed in the section specified in this directive. Uninitialized global variables are placed in the specified section, but const-qualified global variables and auto variables are excluded from this processing.

When this pragma directive is not used, uninitialized global variables are placed in the lwm\_comm section.

#### #pragma unroll

* Format

#pragma unroll [(]<value>[)]

<value>: 0 ~ 32

* Description

Loop expansion is performed on the loop statement immediately below at a multiple specified in the numerical value. At this time, the upper limit of loop expansion specified by the option -Ounroll is ignored.

The specifiable numerical values are 0 to 32.

If any other value is specified, an error occurs. When the number 0 is specified, loop expansion is not performed in the same way as the number 1 is specified. It is an error if the numerical value is omitted.

If no loop statement exists immediately below this pragma directive, a warning is output and this pragma directive is ignored.  
#pragma unroll is a requirement for the compiler, and it is not always loop unrolled as required.

This #pragma is available from V3U.

* Example (Example of how to specify #pragma unroll and loop statement)

|  |
| --- |
| void func(void)  {  int i;  # pragma unroll 4 // Because the statement directly below is not a loop statement,  // Output warning and ignore pragma directive  i = 0;  while (i<8) {  ……  ++i;  }  …… |

#### #pragma gwm\_write\_only

* Format

#pragma gwm\_write\_only [(] <function name> [, ...] [)]

* Description

In the function specified by this pragma directive, only writing operation to the GWM area is permitted, and reading operation is not guaranteed.  
Functions that do not specify this pragma directive do not guarantee the operation when writing to the GWM area, but reading is permitted.  
The user needs to divide write processing and read processing for the GWM area, and specify this pragma command for the write processing function. When returning from the call of the function specified in this pragma directive, it is possible to read from the GWM area.  
An error will occur if the same function name is specified more than once in this pragma directive, or if it matches with another designated function of #pragma directive.  
An error occurs if a function call and a #pragma inline\_asm specification function are described in the function specified by this pragma directive. However, descriptions of built-in functions are not considered as errors.

* Example: #pragma gwm\_write\_only specification example

|  |
| --- |
| #pragma gwm\_write\_only func\_w,func1,func2  #pragma inline\_asm func\_asm  int i;  void func\_w(int j)  {  i = j; // in a function with #pragma gwm\_write\_only directive  // writing to the GWM area is possible, reading is not possible  }  int func\_r(void)  {  return i; // despite the function with #pragma gwm\_write\_only directive  // writing to the GWM area is impossible, reading is possible  void func1(void)  {  ++i; // despite the function with #pragma gwm\_write\_only directive  // operation is not guaranteed because it reads from the GWM area  }  static void func\_asm(void)  {  Assembler instruction  }  void func2(int j, int k)  {  // func2 is for #pragma gwm\_write\_only directive function  func(); // Description of function call is an error  func\_asm(); // Inline\_asm specified function error is an error  \_\_max(j, k); // Description of built-in functions is possible  } |

### Built-in functions

#### List of built-in functions

Table 1.7 Built-in functions using only basic data type

| Assembler Instruction | Functionality | Built-in Function | Remarks (Error Specification, etc.) |
| --- | --- | --- | --- |
| abs D0, a | Absolute value | int a;  int \_\_abs(a); |  |
| actst data | ACTST for integer types argument | int data;  void \_\_actst(data); |  |
| actst data | ACTST for floating-point type argument | float data;  void \_\_actstf(data); |  |
| adddbl D0, s0, s1  or  adddblu.i D0, s0, s1 | ADDDBL | unsigned long long s0;  unsigned int s1;  unsigned long long \_\_adddbl(s0, s1); |  |
| mov r9, a \*  mov D0, r9  \* May be a transfer instruction other than mov | Cosine of π × argument | float a;  float \_\_cospif(a); |  |
| cnt0 D0, a | Number of 0 bits | int a;  int \_\_count0(a); |  |
| cnt1 D0, a | Number of 1 bits | int a;  int \_\_count1(a); |  |
| mov r11, a \*  mov D0, r11  \* May be a transfer instruction other than mov | A power of 2 | float a;  float \_\_exp2f(a); |  |
| fabs D0, a | Absolute value | float a;  float \_\_fabsf(a); |  |
| fflr D0, a | Floor | float a;  float \_\_floorf(a); |  |
| fmax D0, a, b  or  fmaxu.f D0, a, b | Maximum value | float a, b;  float \_\_fmaxf(a, b); |  |
| fmin D0, a, b  or  fminu.f D0, a, b | Minimum value | float a, b;  float \_\_fminf(a, b); |  |
| ffrc D0, a | Fraction | float a;  float \_\_frac(a); |  |
| ldci D0, 0 | Acquisition of core ID | int \_\_getCoreID(void); |  |
| ldrci D0, x, y  or  ldrxy D0, x, y | Referring to image data | int x, y;  int \_\_getSrc(x, y); |  |
| fldrci D0, x, y  or  fldrxy D0, x, y | Referring to image data | int x, y;  float \_\_getSrcf(x, y); |  |
| ldci D0, 1 | Acquisition of thread ID | int \_\_getThreadID(void); |  |
| mov D0, r0 \*  \* r0 is used as an operand of instructions other than mov in some cases. | Referring to X coordinate | int \_\_getX(void); |  |
| mov D0, r1 \*  \* r1 is used as an operand of instructions other than mov in some cases. | Referring to Y coordinate | int \_\_getY(void); |  |
| int | INT | void \_\_int(void); |  |
| llrot D0, s0, s1 | LLROT | unsigned long long s0;  int s1;  unsigned long long \_\_llrot(s0, s1); |  |
| mov r12, a \*  mov D0, r12  \* May be a transfer instruction other than mov | Log | float a;  float \_\_log2f(a); |  |
| macs d0, s0, s1 | MACS | long long d0;  int s0, s1;  long long \_\_macs(d0, s0, s1); |  |
| maxs D0, a, b  or  maxsu.i D0, a, b | Maximum value | int a, b;  int \_\_max(a, b); |  |
| fmp D0, a, b | Intermediate value | float a, b;  float \_\_middlef(a, b); |  |
| mins D0, a, b  or  minsu.i D0, a, b | Minimum value | int a, b;  int \_\_min(a, b); |  |
| mulls d0, s0, s1  or  mullsu.i d0, s0, s1 | MULLS | int s0, s1;  long long \_\_mulls(s0, s1); |  |
| ncscs d0, s0, s1 | NCSCS | int d0, s0, s1;  int \_\_ncscs(d0, s0, s1); |  |
| movi r16, [data(1-16bit)]  sethi r16, data  or  movi r16, [data(1-16bit)] | no uniform load | int data;  int \_\_no\_uniform(data) |  |
| movi r16, [data(1-16bit)]  sethi r16, data  or  movi r16, [data(1-16bit)] | no uniform load | float data;  float \_\_no\_uniformf(data) |  |
| movi r16, [data(1-16bit)]  sethi r16, [data(1-32bit)]  movi r17,[data(33-48bit)]  sethi r17,[data(33-64bit]]  or  movi r16, [data(1-16bit)]  movi r17,[data(33-64bit)] | no uniform load | long long data;  long long \_\_no\_uniforml(data) |  |
| nop | NOP | void \_\_nop(void); |  |
| packb0 D0, s0, s1 | PACKB0 | unsigned long long s0, s1;  unsigned int \_\_packb0(s0, s1); |  |
| packb1 D0, s0, s1 | PACKB1 | unsigned long long s0, s1;  unsigned int \_\_packb1(s0, s1); |  |
| packb2 D0, s0, s1 | PACKB2 | unsigned long long s0, s1;  unsigned int \_\_packb2(s0, s1); |  |
| packb3 D0, s0, s1 | PACKB3 | unsigned long long s0, s1;  unsigned int \_\_packb3(s0, s1); |  |
| packh0 D0, s0, s1 | PACKH0 | unsigned int s0, s1;  unsigned int \_\_packh0(s0, s1); |  |
| packh1 D0, s0, s1 | PACKH1 | unsigned int s0, s1;  unsigned int \_\_packh1(s0, s1); |  |
| mov r14, a \*  mov D0, r14  \* May be a transfer instruction other than mov | Reciprocal operation(RCP0) | float a;  float \_\_rcp0f(float a); |  |
| mov r15, a \*  mov D0, r15  \* May be a transfer instruction other than mov | Reciprocal operation(RCP1) | float a;  float \_\_rcp1f(float a); |  |
| scscs d0, s0, s1 | SCSCS | int d0, s0, s1;  int \_\_scscs(d0, s0, s1); |  |
| strci value, x, y  or  strxy value, x, y | Definition of image data | int x, y, value;  void \_\_setDst(x, y, value); |  |
| fstrci value, x, y  or  fstrxy value, x, y | Definition of image data | int x, y;  float value;  void \_\_setDstf(x, y, value); |  |
| stci cr9, id | Specifying ID of Dst image | short id;  void \_\_setDstImageID(id); | Argument “id” is only an integer constant (0 to 7). If a variable is written, an error occurs. If a value out of the range is written, a warning occurs. |
| stci cr16, mode | Specifying rounding mode for operation | short mode;  void \_\_setRoundMode(mode); | Argument “mode” is only an integer constant (0 or 1). If a variable is written, an error occurs. If a value out of the range is written, a warning occurs. |
| stci cr17, mode | Specifying rounding mode for type conversion | short mode;  void \_\_setRoundModeForConvert(mode); | Argument “mode” is only an integer constant (0 or 1). If a variable is written, an error occurs. If a value out of the range is written, a warning occurs. |
| stci cr18, mode | Specifying rounding mode for executing FFRC and FFLR instructions | short mode;  void \_\_setRoundModeForFloor(mode); | Argument “mode” is only an integer constant (0 or 1). If a variable is written, an error occurs. If a value out of the range is written, a warning occurs. |
| sbocrd data, x, y | Integer output to SBO | signed char x, y;  int data;  void \_\_setSBO(x, y, data); | Argument “x,y” is only a signed 8-bit integer constant. If a variable is written, an error occurs. If a value out of the range is written, a warning occurs. |
| sbocrd data, x, y | Floating-point number output to SBO | signed char x, y;  float data;  void \_\_setSBOf(x, y, data); | Argument “x,y” is only a signed 8-bit integer constant. If a variable is written, an error occurs. If a value out of the range is written, a warning occurs. |
| stci cr10, mask | SBO output information | short mask;  void \_\_setSBOMask(mask); | Argument “mask” is only an integer constant (0 to 7). If a variable is written, an error occurs. If a value out of the range is written, a warning occurs. |
| stci cr8, id | Specifying ID of Src image | short id;  void \_\_setSrcImageID(id); | Argument “id” is only an integer constant (0 to 7). If a variable is written, an error occurs. If a value out of the range is written, a warning occurs. |
| mov r0, x \*  \* May be a transfer instruction other than mov | Definition of X coordinate | int x;  void \_\_setX(x); |  |
| mov r1, y \*  \* May be a transfer instruction other than mov | Definition of Y coordinate | int y;  void \_\_setY(y); |  |
| mov r8, a \*  mov D0, r8  \* May be a transfer instruction other than mov | Sine of π× argument | float a;  float \_\_sinpif(a); |  |
| mov r13, a \*  mov D0, r13  \* May be a transfer instruction other than mov  When division with a divisor of \_\_sqrtf() is described, transfer to r10 is generated. | Square root | float a;  float \_\_sqrtf(a); |  |
| syncg | SYNCG | void \_\_syncg(void); |  |
| syncp mode | SYNCP | signed char mode;  void \_\_syncp(mode); | Argument “mode” is only an integer constant (1 to 3). If a variable is written, an error occurs. If a value out of the range is written, a warning occurs. |
| trap | TRAP | void \_\_trap(void); |  |
| mov r16, s0  mov r17, s1  \* Example of pair register r16r17 as a transfer destination  \* The instruction sequence may not be as shown depending on optimization. | Concatenation of data | unsigned int s0, s1;  unsigned long long \_\_vconcatenate(s0, s1); |  |
| waitdma | WAITDMA | void \_\_waitdma(void); |  |

Table 1.8 Built-in functions using extended data type (vector type)

| Assembler Instruction | Functionality | Built-in Function | Remarks (Error Specification, etc.) |
| --- | --- | --- | --- |
| asm D0, s0, s1 | ASM | \_\_int32x2 s0, s1;  \_\_int32x2 \_\_asmul(s0, s1); | Same as \_\_asm\_s32x2. |
| csuss d0, s0, s1 | CSUSS | \_\_int8x4 d0, s0, s1;  \_\_int8x4 \_\_csuss(d0, s0, s1); | Same as \_\_csus\_s8x4. |
| csusu d0, s0, s1 | CSUSU | \_\_uint8x4 d0, s0, s1;  \_\_uint8x4 \_\_csusu(d0, s0, s1); | Same as \_\_csus\_u8x4. |
| gaddbs d0, s0, s1 | GADDBS | int d0;  \_\_int8x8 s0, s1;  int \_\_gaddbs(d0, s0, s1); | Same as \_\_gadd\_s8x8. |
| gaddbu d0, s0, s1 | GADDBU | int d0;  \_\_uint8x8 s0, s1;  int \_\_gaddbu(d0, s0, s1); | Same as \_\_gadd\_u8x8. |
| gaddhs d0, s0, s1 | GADDHS | int d0;  \_\_int16x4 s0, s1;  int \_\_gaddhs(d0, s0, s1); | Same as \_\_gadd\_s16x4. |
| gaddhu d0, s0, s1 | GADDHU | int d0;  \_\_uint16x4 s0, s1;  int \_\_gaddhu(d0, s0, s1); | Same as \_\_gadd\_u16x4. |
| gmadbs d0, s0, s1 | GMADBS | int d0;  \_\_int8x8 s0, s1;  int \_\_gmadbs(d0, s0, s1); | Same as \_\_gmad\_s8x8. |
| gmadbu d0, s0, s1 | GMADBU | int d0;  \_\_uint8x8 s0, s1;  int \_\_gmadbu(d0, s0, s1); | Same as \_\_gmad\_u8x8. |
| gmadhs d0, s0, s1 | GMADHS | int d0;  \_\_int16x4 s0, s1;  int \_\_gmadhs(d0, s0, s1); | Same as \_\_gmad\_s16x4. |
| gmadhu d0, s0, s1 | GMADHU | int d0;  \_\_uint16x4 s0, s1;  int \_\_gmadhu(d0, s0, s1); | Same as \_\_gmad\_u16x4. |
| gsadbs d0, s0, s1 | GSADBS | int d0;  \_\_int8x8 s0, s1;  int \_\_gsadbs(d0, s0, s1); | Same as \_\_gsad\_s8x8. |
| gsadbu d0, s0, s1 | GSADBU | int d0;  \_\_uint8x8 s0, s1;  int \_\_gsadbu(d0, s0, s1); | Same as \_\_gsad\_u8x8. |
| gsadhs d0, s0, s1 | GSADHS | int d0;  \_\_int16x4 s0, s1;  int \_\_gsadhs(d0, s0, s1); | Same as \_\_gsad\_s16x4. |
| gsadhu d0, s0, s1 | GSADHU | int d0;  \_\_uint16x4 s0, s1;  int \_\_gsadhu(d0, s0, s1); | Same as \_\_gsad\_u16x4. |
| gsubbs d0, s0, s1 | GSUBBS | int d0;  \_\_int8x8 s0, s1;  int \_\_gsubbs(d0, s0, s1); | Same as \_\_gsub\_s8x8. |
| gsubbu d0, s0, s1 | GSUBBU | int d0;  \_\_uint8x8 s0, s1;  int \_\_gsubbu(d0, s0, s1); | Same as \_\_gsub\_u8x8. |
| gsubhs d0, s0, s1 | GSUBHS | int d0;  \_\_int16x4 s0, s1;  int \_\_gsubhs(d0, s0, s1); | Same as \_\_gsub\_s16x4. |
| gsubhu d0, s0, s1 | GSUBHU | int d0;  \_\_uint16x4 s0, s1;  int \_\_gsubhu(d0, s0, s1); | Same as \_\_gsub\_u16x4. |
| llrot D0, s0, s1 | LLROT | \_\_int16x4 s0;  int s1;  \_\_int16x4 \_\_llrot\_s16x4(s0, s1); |  |
| llrot D0, s0, s1 | LLROT | \_\_int32x2 s0;  int s1;  \_\_int32x2 \_\_llrot\_s32x2(s0, s1); |  |
| llrot D0, s0, s1 | LLROT | \_\_int8x8 s0;  int s1;  \_\_int8x8 \_\_llrot\_s8x8(s0, s1); |  |
| llrot D0, s0, s1 | LLROT | \_\_uint16x4 s0;  int s1;  \_\_uint16x4 \_\_llrot\_u16x4(s0, s1); |  |
| llrot D0, s0, s1 | LLROT | \_\_uint8x8 s0;  int s1;  \_\_uint8x8 \_\_llrot\_u8x8(s0, s1); |  |
| packh0 D0, s0, s1 | PACKH0 | \_\_int16x2 s0, s1;  \_\_int16x2 \_\_pack\_s16x2\_p0(s0, s1); |  |
| packh1 D0, s0, s1 | PACKH1 | \_\_int16x2 s0, s1;  \_\_int16x2 \_\_pack\_s16x2\_p1(s0, s1); |  |
| packb0 D0, s0, s1 | PACKB0 | \_\_int8x8 s0, s1;  \_\_int8x4 \_\_pack\_s8x4\_p0(s0, s1); |  |
| packb1 D0, s0, s1 | PACKB1 | \_\_int8x8 s0, s1;  \_\_int8x4 \_\_pack\_s8x4\_p1(s0, s1); |  |
| packb2 D0, s0, s1 | PACKB2 | \_\_int8x8 s0, s1;  \_\_int8x4 \_\_pack\_s8x4\_p2(s0, s1); |  |
| packb3 D0, s0, s1 | PACKB3 | \_\_int8x8 s0, s1;  \_\_int8x4 \_\_pack\_s8x4\_p3(s0, s1); |  |
| packh0 D0, s0, s1 | PACKH0 | \_\_uint16x2 s0, s1;  \_\_uint16x2 \_\_pack\_u16x2\_p0(s0, s1); |  |
| packh1 D0, s0, s1 | PACKH1 | \_\_uint16x2 s0, s1;  \_\_uint16x2 \_\_pack\_u16x2\_p1(s0, s1); |  |
| packb0 D0, s0, s1 | PACKB0 | \_\_uint8x8 s0, s1;  \_\_uint8x4 \_\_pack\_u8x4\_p0(s0, s1); |  |
| packb1 D0, s0, s1 | PACKB1 | \_\_uint8x8 s0, s1;  \_\_uint8x4 \_\_pack\_u8x4\_p1(s0, s1); |  |
| packb2 D0, s0, s1 | PACKB2 | \_\_uint8x8 s0, s1;  \_\_uint8x4 \_\_pack\_u8x4\_p2(s0, s1); |  |
| packb3 D0, s0, s1 | PACKB3 | \_\_uint8x8 s0, s1;  \_\_uint8x4 \_\_pack\_u8x4\_p3(s0, s1); |  |
| shli D0, s0, i | Moving vector factor using the shift instruction | \_\_int8x4 s0;  int i;  \_\_int8x4 \_\_shl\_s8x4(s0, i); |  |
| shli D0, s0, i | Moving vector factor using the shift instruction | \_\_uint8x4 s0;  int i;  \_\_uint8x4 \_\_shl\_u8x4(s0, i); |  |
| shrli D0, s0, i | Moving vector factor using the shift instruction | \_\_int8x4 s0;  int i;  \_\_int8x4 \_\_shr\_s8x4(s0, i); |  |
| shrli D0, s0, i | Moving vector factor using the shift instruction | \_\_uint8x4 s0;  int i;  \_\_uint8x4 \_\_shr\_u8x4(s0, i); |  |
| mov r16, v0  mov r17, v1  \* Example of pair register r16r17 as a transfer destination  \* The instruction sequence may not be as shown depending on optimization. | Concatenation of vectors | \_\_int16x2 v0, v1;  \_\_int16x4 \_\_vconcatenate\_s16x4(v0, v1); |  |
| mov r16, v0  mov r17, v1  \* Example of pair register r16r17 as a transfer destination  \* The instruction sequence may not be as shown depending on optimization. | Concatenation of vectors | \_\_int8x4 v0, v1;  \_\_int8x8 \_\_vconcatenate\_s8x8(v0, v1); |  |
| mov r16, v0  mov r17, v1  \* Example of pair register r16r17 as a transfer destination  \* The instruction sequence may not be as shown depending on optimization. | Concatenation of vectors | \_\_uint16x2 v0, v1;  \_\_uint16x4 \_\_vconcatenate\_u16x4(v0, v1); |  |
| mov r16, v0  mov r17, v1  \* Example of pair register r16r17 as a transfer destination  \* The instruction sequence may not be as shown depending on optimization. | Concatenation of vectors | \_\_uint8x4 v0, v1;  \_\_uint8x8 \_\_vconcatenate\_u8x8(v0, v1); |  |

Table 1.9 Built-in functions using only basic data type only for V3U

| Assembler Instruction | Functionality | Built-in Function | Remarks (Error Specification, etc.) |
| --- | --- | --- | --- |
| adds d0, s0, s1  or  addsi d0, s0, s1  or  addsu d0, s0, s1 | Add with Saturation | int s0, s1;  int \_\_adds(s0, s1); |  |
| and64 d0, s0, s1 | AND64 | unsigned long long s0, s1;  unsigned long long \_\_and64(s0, s1); |  |
| btst d0, data, bit  or  btsti d0, data, bit  or  btstu d0, bit, data | Bit Test | unsigned int data, bit;  int \_\_btst(data, bit); | When an argument “bit” is integer constant, it’s range is from 0 to 31. If a value out of the range is written, an error occurs. |
| dmawaits i | DMAWAITS | int i;  void \_\_dmawaits(i); | Argument “i” is only an integer constant (0 to 255). If a value out of the range is written, an error occurs. |
| eor64 d0, s0, s1 | EOR64 | unsigned long long s0, s1;  unsigned long long \_\_eor64(s0, s1); |  |
| inv64 d0, s0 | INV64 | unsigned long long s0;  unsigned long long \_\_inv64(s0); |  |
| lmbd d0, s0 | LMBD | unsigned int s0;  unsigned int \_\_lmbd(s0); |  |
| muls d0, s0, s1  or  mulsu d0, s0, s1 | Multiply with Saturation | int s0, s1;  int \_\_muls(s0, s1); |  |
| or64 d0, s0, s1 | OR64 | unsigned long long s0, s1;  unsigned long long \_\_or64(s0, s1); |  |
| plsb8 d0, s0, s1 | PLSB8 | unsigned int s0, s1;  unsigned int \_\_plsb8(s0, s1); |  |
| plsb16 d0, s0, s1 | PLSB16 | unsigned int s0, s1;  unsigned int \_\_plsb16(s0, s1); |  |
| rmbd d0, s0 | RMBD | unsigned int s0;  unsigned int \_\_rmbd(s0); |  |
| shuf d0, s0, s1 | Shuffle | unsigned long long d0, s0, s1;  unsigned long long \_\_shuf(d0, s0, s1); |  |
| subs d0, s0, s1  or  subsru d0, s0, s1  or  subsur d0, s1, s0 | Subtract with Saturation | int s0, s1;  int \_\_subs(s0, s1); |  |
| uplsb8 d0, s0 | UPLSB8 | unsigned int d0, s0;  unsigned int \_\_uplsb8(d0, s0); |  |
| uplsb16 d0, s0 | UPLSB16 | unsigned int d0, s0;  unsigned int \_\_uplsb16(d0, s0); |  |

Table 1.10 Built-in functions using extended data type (vector type) only for V3U

| Assembler Instruction | Functionality | Built-in Function | Remarks (Error Specification, etc.) |
| --- | --- | --- | --- |
| abs8 d0, s0 | ABS8 | \_\_int8x8 s0;  \_\_uint8x8 \_\_abs8(s0); |  |
| abs16 d0, s0 | ABS16 | \_\_int16x4 s0;  \_\_uint16x4 \_\_abs16(s0); |  |
| addss8 d0, s0, s1 | ADDSS8 | \_\_int8x8 s0, s1;  \_\_int8x8 \_\_addss8(s0, s1); |  |
| addss16 d0, s0, s1 | ADDSS16 | \_\_int16x4 s0, s1  \_\_int16x4 \_\_addss16(s0, s1); |  |
| addus8 d0, s0, s1 | ADDUS8 | \_\_uint8x8 s0, s1;  \_\_uint8x8 \_\_addus8(s0, s1); |  |
| addus16 d0, s0, s1 | ADDUS16 | \_\_uint16x4 s0, s1;  \_\_uint16x4 \_\_addus16(s0, s1); |  |
| cmp.eq8 d0, s0, s1 | CMP.EQ8 | \_\_int8x8 s0, s1;  \_\_uint8x8 \_\_cmp\_eq8(s0, s1); |  |
| cmp.eq16 d0, s0, s1 | CMP.EQ16 | \_\_int16x4 s0, s1;  \_\_uint16x4 \_\_cmp\_eq16(s0, s1); |  |
| cmp.neq8 d0, s0, s1 | CMP.NEQ8 | \_\_int8x8 s0, s1;  \_\_uint8x8 \_\_cmp\_ne8(s0, s1); |  |
| cmp.neq16 d0, s0, s1 | CMP.NEQ16 | \_\_int16x4 s0, s1;  \_\_uint16x4 \_\_cmp\_ne16(s0, s1); |  |
| cmp.sle8 d0, s0, s1 | CMP.SLE8 | \_\_int8x8 s0, s1;  \_\_uint8x8 \_\_cmp\_sle8(s0, s1); |  |
| cmp.sle16 d0, s0, s1 | CMP.SLE16 | \_\_int16x4 s0, s1;  \_\_uint16x4 \_\_cmp\_sle16(s0, s1); |  |
| cmp.slt8 d0, s0, s1 | CMP.SLT8 | \_\_int8x8 s0, s1;  \_\_uint8x8 \_\_cmp\_slt8(s0, s1); |  |
| cmp.slt16 d0, s0, s1 | CMP.SLT16 | \_\_int16x4 s0, s1;  \_\_uint16x4 \_\_cmp\_slt16(s0, s1); |  |
| cmp.ule8 d0, s0, s1 | CMP.ULE8 | \_\_uint8x8 s0, s1;  \_\_uint8x8 \_\_cmp\_ule8(s0, s1); |  |
| cmp.ule16 d0, s0, s1 | CMP.ULE16 | \_\_uint16x4 s0, s1;  \_\_uint16x4 \_\_cmp\_ule16(s0, s1); |  |
| cmp.ult8 d0, s0, s1 | CMP.ULT8 | \_\_uint8x8 s0, s1;  \_\_uint8x8 \_\_cmp\_ult8(s0, s1); |  |
| cmp.ult16 d0, s0, s1 | CMP.ULT16 | \_\_uint16x4 s0, s1;  \_\_uint16x4 \_\_cmp\_ult16(s0, s1); |  |
| cmv8 d0, s0, s1 | CMV8 | \_\_uint8x8 d0;  unsigned long long s0;  unsigned int s1;  \_\_uint8x8 \_\_cmv8(d0, s0, s1); |  |
| cmv16 d0, s0, s1 | CMV16 | \_\_uint16x4 d0;  unsigned long long s0;  unsigned int s1;  \_\_uint16x4 \_\_cmv16(d0, s0, s1); |  |
| gmadbsl d0, s0, s1 | GMADBSL | long long d0;  \_\_int8x8 s0, s1;  long long \_\_gmadbsl(d0, s0, s1); |  |
| gmadbul d0, s0, s1 | GMADBUL | unsigned long long d0;  \_\_uint8x8 s0, s1;  unsigned long long \_\_gmadbul(d0, s0, s1); |  |
| gmadhsl d0, s0, s1 | GMADHSL | long long d0;  \_\_int16x4 s0, s1;  long long \_\_gmadhsl(d0, s0, s1); |  |
| gmadhul d0, s0, s1 | GMADHUL | unsigned long long d0;  \_\_uint16x4 s0, s1;  unsigned long long \_\_gmadhul(d0, s0, s1); |  |
| maxs8 d0, s0, s1 | MAXS8 | \_\_int8x8 s0, s1;  \_\_int8x8 \_\_maxs8(s0, s1); |  |
| maxs16 d0, s0, s1 | MAXS16 | \_\_int16x4 s0, s1;  \_\_int16x4 \_\_maxs16(s0, s1); |  |
| maxu8 d0, s0, s1 | MAXU8 | \_\_uint8x8 s0, s1;  \_\_uint8x8 \_\_maxu8(s0, s1); |  |
| maxu16 d0, s0, s1 | MAXU16 | \_\_uint16x4 s0, s1;  \_\_uint16x4 \_\_maxu16(s0, s1); |  |
| mins8 d0, s0, s1 | MINS8 | \_\_int8x8 s0, s1;  \_\_int8x8 \_\_mins8(s0, s1); |  |
| mins16 d0, s0, s1 | MINS16 | \_\_int16x4 s0, s1;  \_\_int16x4 \_\_mins16(s0, s1); |  |
| minu8 d0, s0, s1 | MINU8 | \_\_uint8x8 s0, s1;  \_\_uint8x8 \_\_minu8(s0, s1); |  |
| minu16 d0, s0, s1 | MINU16 | \_\_uint16x4 s0, s1;  \_\_uint16x4 \_\_minu16(s0, s1); |  |
| mulss8 d0, s0, s1 | MULSS8 | \_\_int8x8 s0, s1;  \_\_int8x8 \_\_mulss8(s0, s1); |  |
| mulss16 d0, s0, s1 | MULSS16 | \_\_int16x4 s0, s1;  \_\_int16x4 \_\_mulss16(s0, s1); |  |
| mulls8 d0, s0, s1 | MULLS8 | \_\_int8x4 s0, s1;  \_\_int16x4 \_\_mulls8(s0, s1); |  |
| mulls16 d0, s0, s1 | MULLS16 | \_\_int16x2 s0, s1;  \_\_int32x2 \_\_mulls16(s0, s1); |  |
| mullu8 d0, s0, s1 | MULLU8 | \_\_uint8x4 s0, s1;  \_\_uint16x4 \_\_mullu8(s0, s1); |  |
| mullu16 d0, s0, s1 | MULLU16 | \_\_uint16x2 s0, s1;  \_\_uint32x2 \_\_mullu16(s0, s1); |  |
| mulus8 d0, s0, s1 | MULUS8 | \_\_uint8x8 s0, s1;  \_\_uint8x8 \_\_mulus8(s0, s1); |  |
| mulus16 d0, s0, s1 | MULUS16 | \_\_uint16x4 s0, s1;  \_\_uint16x4 \_\_mulus16(s0, s1); |  |
| pcmpagt d0, s0, s1 | PCMPAGT | \_\_int32x2 s0, s1;  int \_\_pcmpagt(s0, s1); |  |
| pcmpalt d0, s0, s1 | PCMPALT | \_\_int32x2 s0, s1;  int \_\_pcmpalt(s0, s1); |  |
| pcmpaeq d0, s0, s1 | PCMPAEQ | \_\_int32x2 s0, s1;  int \_\_pcmpaeq(s0, s1); |  |
| pcmpogt d0, s0, s1 | PCMPOGT | \_\_int32x2 s0, s1;  int \_\_pcmpogt(s0, s1); |  |
| pcmpolt d0, s0, s1 | PCMPOLT | \_\_int32x2 s0, s1;  int \_\_pcmpolt(s0, s1); |  |
| pcmpoeq d0, s0, s1 | PCMPOEQ | \_\_int32x2 s0, s1;  int \_\_pcmpoeq(s0, s1); |  |
| rxch d0, s0 | RXCH | \_\_int32x2 s0;  \_\_int32x2 \_\_rxch(s0); |  |
| sha8 d0, s0, s1 | SHA8 | \_\_int8x8 s0, s1;  \_\_int8x8 \_\_sha8(s0, s1); |  |
| sha16 d0, s0, s1 | SHA16 | \_\_int16x4 s0, s1;  \_\_int16x4 \_\_sha16(s0, s1); |  |
| shai8 d0, s0, i | SHAI8 | \_\_int8x8 s0;  int i;  \_\_int8x8 \_\_shai8(s0, i); | Argument “i” is only an integer constant (-7 to 7). If a value out of the range is written, an error occurs. |
| shai16 d0, s0, i | SHAI16 | \_\_int16x4 s0;  int i;  \_\_int16x4 \_\_shai16(s0, i); | Argument “i” is only an integer constant (-15 to 15). If a value out of the range is written, an error occurs. |
| shl8 d0, s0, s1 | SHL8 | \_\_uint8x8 s0;  \_\_int8x8 s1;  \_\_uint8x8 \_\_shl8(s0, s1); |  |
| shl16 d0, s0, s1 | SHL16 | \_\_uint16x4 s0;  \_\_int16x4 s1;  \_\_uint16x4 \_\_shl16(s0, s1); |  |
| shli8 d0, s0, i | SHLI8 | \_\_uint8x8 s0;  int i;  \_\_uint8x8 \_\_shli8(s0, i) | Argument “i” is only an integer constant (-7 to 7). If a value out of the range is written, an error occurs. |
| shli16 d0, s0, i | SHLI16 | \_\_uint16x4 s0;  int i;  \_\_uint16x4 \_\_shli16(s0, i); | Argument “i” is only an integer constant (-15 to 15). If a value out of the range is written, an error occurs. |
| subss8 d0, s0, s1 | SUBSS8 | \_\_int8x8 s0, s1;  \_\_int8x8 \_\_subss8(s0, s1); |  |
| subss16 d0, s0, s1 | SUBSS16 | \_\_int16x4 s0, s1;  \_\_int16x4 \_\_subss16(s0, s1); |  |
| subus8 d0, s0, s1 | SUBUS8 | \_\_uint8x8 s0, s1;  \_\_uint8x8 \_\_subus8(s0, s1); |  |
| subus16 d0, s0, s1 | SUBUS16 | \_\_uint16x4 s0, s1;  \_\_uint16x4 \_\_subus16(s0, s1); |  |

#### Notes on describing built-in functions

When describing a built-in function, the sequence of the built-in function and the C language source processing placed before and after the built-in function may not be guaranteed in the instruction sequence after compiling.

In the case of the following built-in functions, the compiler interprets them as "a memory read instruction" and determines the order of instructions. (For built-in functions having dummy arguments or return values, the dependence of their data is also considered.)

\_\_getSrc, \_\_getSrcf, \_\_getX, \_\_getY

For this reason, with respect to memory write and function call, the C language source description processing order matches the instruction output order. However, with respect to processing for which mutual dependence and adverse reaction need not be considered (such as operation between registers), the C language source description processing order may not match the instruction output order.

For other built-in functions that have \_\_getCoreID, \_\_getThreadID, or both dummy arguments and return values, the compiler interprets them as "an instruction without memory access" and determines the order of instructions. The compiler determines the order of instructions based on the dependence of data of dummy arguments or return values. If there is no effect on the dependence of data, the call processing order of functions described in the C language source and the processing order of built-in functions may exchange in the instruction sequence.

For other built-in functions, the compiler interprets them as "an instruction with adverse reaction that may update memory data" and determines the order of instructions. (For built-in functions having dummy arguments or return values, the dependence of their data is also considered.) For this reason, with respect to memory read/write and function call, the C language source description processing order matches the instruction output order. However, with respect to processing for which mutual dependence and adverse reaction need not be considered (such as operation between registers), the C language source description processing order may not match the instruction output order.

* Example 1 where processing sequence is exchanged

C language source:

int x;

void f(int a, int b){

int tmp = a+b;

\_\_nop();

x = tmp;

}

Instruction sequence after compiling:

\_f:

nop

add r16, r16, r17

movi r17, \_x

sethi r17, \_x

sti r16, r17, 0x00000000

brar r31

\* The Add instruction is moved to after nop because there is no need to consider effects of adverse reaction of the Add instruction.

* Example 2 where processing sequence is exchanged

C language source:

float f(float a, float b, float c){

float x = a;

\_\_setRoundMode(1);

x += b;

\_\_setRoundMode(0);

x += c;

return x;

}

Instruction sequence after compiling:

\_f:

stci CR16, 0x00000001

stci CR16, 0x00000000

fadd r16, r16, r17

fadd r16, r16, r18

brar r31

\* There is no dependence between Add instruction and built-in function. Even if a built-in function is written in order to switch the rounding mode in each operation, the instruction sequence is not guaranteed.

## Unspecified behavior

|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Standard chapter number** | **Item** | **Specification** |
| 1 | 5.1.2 | Execution environment-static storage initialization | Static data is output at compile time as data sections. |
| 2 | 5.1.2.2.3 | Program termination processing-termination status where the main function is returned to the host environment | If the return type of the main function does not match with int, return the value with the declaration type. |
| 3 | 5.2.2 | Textual meaning - end position of line | The behavior of the display when the display character is written when the current display position is at the end of the line depends on the display. |
| 4 | 5.2.2 | Meaning of character display-Back character (\ b) | The behavior of the display when the backward character is written when the current display position is at the first position of the line depends on the display. |
| 5 | 5.2.2 | Lettering Meaning - Horizontal Tab (\ t) | The behavior of the display when the current display position is at the last defined horizontal tab position or if the horizontal tab is written if the position is past is dependent on the display. |
| 6 | 5.2.2 | Character Meaning - Vertical Tab (\ v) | The behavior of the display when the vertical character is written if the display position is at or past the last defined vertical tab position depends on the display. |
| 7 | 5.2.4.1 | Character display-number of characters of source extended character not corresponding to international character name | International character names are not supported. |
| 8 | 6.2.6 | Type - floating point and vector | . IEEE 754 compliant for floating point type. However, double precision floating point type is not supported. If double type or long double type is described, it is treated as float type.  . Supports vector type as an extended language specification. |
| 9 | 6.2.6.1 | Padding byte value | Padding values ​​when storing values ​​in structures / unions are undefined. |
| 10 | 6.2.6.1 | Union member access | When accessing a member of a union object using a member other than the union member that stored the value last, the internal representation of data conforms to the type to be accessed. |
| 11 | 6.2.6.1 | Representation when storing in object | Described in "Internal representation and area of ​​data". |
| 12 | 6.2.6.2 | Integer representation - the value of any padding bit | Described in "Internal representation and area of ​​data". |
| 13 | 6.2.6.2 | Integer representation - negative 0 | Since a signed integer type is expressed by 2's complement, negative 0 is not supported. |
| 14 | 6.4.5 | Handling of two string literals | String literals are not supported. |
| 15 | 6.5 | The order in which subexpressions are evaluated, and the order in which side effects occur | The order in which subexpressions are evaluated and the order in which side effects occur are unspecified. |
| 16 | 6.5.2.2 | Function call - evaluation order of function directives, actual arguments and subexpressions within actual arguments | The evaluation order of function directives, actual arguments, and subexpressions within actual arguments in function calls is unspecified. |
| 17 | 6.5.2.5 | Compound literals - the order in which side effects occur in the initialization list expression | The order in which side effects occur is in the order of the elements. |
| 18 | 6.5.16 | Assignment operator - the order in which the operands are evaluated | The order in which the assignment operator's operands are evaluated is unspecified. |
| 19 | 6.7.2.1 | Alignment of addressable storage units allocated to hold bit fields | Bit fields are not supported. |
| 20 | 6.7.4 | How to call inline function | The inline function specifier is not supported. |
| 21 | 6.7.5.2 | Evaluation of variable length array type expressions | Variable length array type is not supported. |
| 22 | 6.7.8 | Order of occurrence of side effects in initializer list expressions | The order in which side effects occur is in the order of elements (members).  However, when optimization is performed, it is not specified. |
| 23 | 6.9.1 | Placement of storage for function formal arguments | Allocate to stack and register. The details are described in "Function call specification". |
| 24 | 6.10.3 | Functional macros - nested substitution | If you nest function type macros of the same name in function type macros, they will not be expanded. |
| 25 | 6.10.3.2  6.10.3.3 | # Operator / ## operator - evaluation order | It evaluate forward. |
| 26 | 7.5 | definition of errno | errno is a macro definition. |
| 27 | 7.6.1 | FENV\_ACCESS pragma | # pragma STDC FENV\_ACCESS only performs syntax check and ignores the contents. |
| 28 | 7.6.2.3 | feraiseexcept function - order to generate specified floating point exceptions | The feraiseexcept function is not supported. |
| 29 | 7.12 | definition of math\_errhandling | math\_errhandling is a macro definition. |
| 30 | 7.12.6.4 | frexp functions - result if value is not a floating point number | Return value changes according to the input value.  Input value: Return value  Infinity : 1  NaN : 2 |
| 31 | 7.12.9.5  F. 9.6.5 | lrint / llrint function group-result value | lrint function, lrintl function, llrint function group is not supported. |
| 32 | 7.12.9.7  F. 9.6.5 | lround / llround function group-result value | The lround function, lroundl function and llround function group are not supported. |
| 33 | 7.13 | definition of setjmp | The setjmp function is not supported. |
| 34 | 7.15.1 | Definition of va\_copy and va\_end | va\_copy function and va\_end function are not supported. |
| 35 | 7.19.6.1  7.24.2.1 | a/A conversion specifier-hexadecimal digit before decimal character on output | The fprintf and fwprintf functions are not supported. |
| 36 | 7.19.7.11 | ungetc function - file position indicator value | The ungetc function is not supported. |
| 37 | 7.24.3.10 | ungetwc function - value of file position indicator | The ungetwc function is not supported. |
| 38 | 7.19.9.1 | Details of the value of the second argument stored by the fsetpos function   fgetpos function - details of the value to be stored | The fgetpos function is not supported. |
| 39 | 7.19.9.4 | ftell function-details of the value returned | The ftell function is not supported. |
| 40 | 7.20.1.3  7.24.4.1.1 | Numeric conversion function / strtod function group / wcstod function group - the method of converting the conversion target sequence with a negative sign | The functions strtod, strtof, strtold, wcstod, wcstof and wcstold are not supported. |
| 41 | 7.20.3 | Storage management functions-storage order and adjacency | Dynamic memory management functions are not supported. |
| 42 | 7.20.3 | Storage Management Functions-Amount of Storage | Dynamic memory management functions are not supported. |
| 43 | 7.20.5.1 | bsearch function - return element | The bsearch function is not supported. |
| 44 | 7.20.5.2 | qsort function-order in aligned array | The qsort function is not supported. |
| 45 | 7.23.2.4 | time function-return format of calendar time | The time function is not supported. |
| 46 | 7.23.3.5  7.24.5.1 | Strftime function / wcsftime function-character to store for out-of-range values | The strftime function and wcsftime function are not supported. |
| 47 | 7.24.6.3.2  7.24.6.3.3  7.24.6.4.1  7.24.6.4.2 | mbrtowc Function / wcrtomb Functions / mbsrtowcs Functions / wcsrtombs Functions - conversion state after expression form error | The functions mbrtowc, wcrtomb, mbsrtowcs and wcsrtombs are not supported. |
| 48 | F.4 | Floating-point to integer conversion-value resulting from floating-point exception "invalid" | It is undefined if the floating point exception "invalid" occurs. |
| 49 | F.4 | Converting non-integer floating-point values ​​to integers - whether the floating-point exception "inexact" occurs | The floating point exception "inexact" does not occur. |
| 50 | F.9.3.4 | frexp function group - value of exp at the time of NaN or infinity value | Set 0 to the object pointed by exp in the frexp function group. |
| 51 | F. 9.6.5  F.9.6.7 | lrint function, llrint function, lround function, llround function-return value when the rounded value is out of the range of return type | The llrint function and llround function are not supported. |
| 52 | G.6.1.1  G.6.2.2  G.6.2.3  G.6.2.4  G.6.2.5  G.6.2.6  G.6.3.1  G.6.4.2 | Mathematical functions - Signs of complex results when you specify exception values | Complex types are not supported. |

## Undefined behavior

|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Standard chapter number** | **Item** | **Specification** |
| 1 | 4. | Violation of requirements that the program other than the restrictions of the standard should follow | If the program does not comply with the "... (must)" or "... (do) must not appear" requirements that appear outside the restrictions of the standard, the operation is not guaranteed. |
| 2 | 5.1.1.2 | Source file-end of file | The source file must end with a newline character, but will end normally even if it does not end with a newline character. |
| 3 | 5.1.1.2 | If the source file ends with a newline character, there must not be a back slash before the newline character, but it will end normally even if it does. |
| 4 | 5.1.1.2 | An error occurs if the source file ends in the middle of the preprocessing token or in the middle of the comment. |
| 5 | 5.1.1.2 | Generation of international character names by ## operator | International character names are not supported. |
| 6 | 5.1.2.2.1 | Host environment program | CCIMP is not applicable because it is a free standing environment. |
| 7 | 5.2.1 | Character set | If the source file contains characters that are not in the source basic character set, a message is output. |
| 8 | 5.2.1.2 | Illegal shift state in multibyte characters | Expression format dependent on shift state is not supported. |
| 9 | 5.2.1.2 | Invalid multibyte character | Treat invalid multibyte characters as valid multibyte characters.  However, not only multibyte characters but also character strings cannot be described. |
| 10 | 6.2.2 | Identifier binding | Outputs a message when the same identifier appears in both inner and outer linkages in a translation unit. |
| 11 | 6.2.4 | Object reference out of lifetime | If you refer to an object outside its lifetime, there is no guarantee of operation. |
| 12 | 6.2.4 | Using pointer values ​​to point to out-of-life objects | The value of a pointer that points to an out-of-life object is considered undefined. |
| 13 | 6.2.4  6.7.8  6.8 | Automatic storage period object value used indefinitely | If you do not set the value of the automatic storage object, a warning will be output.   If you reference without setting the value of the automatic storage object, the value is undefined. |
| 14 | 6.2.6.1 | Non-Letter Lvalue Expression-Trap Representation | There is no trap expression. |
| 15 | 6.2.6.1 | Non-Letter lvalue expressions-trap expressions due to side effects | There is no trap expression. |
| 16 | 6.2.6.2 | The result of an operation with an operand that produces a negative 0 | It is the value of the operation type expressed in 2's complement. |
| 17 | 6.2.7 | Adaptation type and synthetic type | All declarations that refer to the same object or function must match. Otherwise, it outputs a message. |
| 18 | 6.3.1.4 | Type conversion-floating point type to integer type | When a floating-point type is converted to an integral type, it is undefined if the value of the integral part cannot be represented by the integral type. |
| 19 | 6.3.1.4 | Type conversion-integer to floating point | When converting an integer type value to a floating point type, if the value of the integer part of the floating point is out of the range of values ​​that can be expressed, it is rounded to a value that can be represented by the floating point type. |
| 20 | 6.3.1.5 | Real floating point type-type conversion between floating point types | Because only single precision floating point types are handled, type conversion between floating point types is not applicable. |
| 21 | 6.3.2.1 | Lvalue-an evaluated lvalue not pointing to an object | Output a warning.   Generate code that points to an object. Operation is not guaranteed. |
| 22 | 6.3.2.1 | Left side value and function directive | If the incomplete type is an lvalue, it is undefined. |
| 23 | 6.3.2.1 | Array type with register storage class | If the array object has register storage class, a warning message will be output. |
| 24 | 6.3.2.2 | Use of void expression value | Using the value of a void expression outputs a message. |
| 25 | 6.3.2.2 | Type conversion for void expressions | Type conversion to void outputs a message. |
| 26 | 6.3.2.3 | Pointer type conversion-pointer type to integer type | If the result of converting a pointer to an integer type exceeds the range that can be represented by that integer type, a warning message will be output. Operation is not guaranteed. |
| 27 | 6.3.2.3 | Pointer type conversion-conversion result without boundary adjustment | Unbounded transformations output a warning message. Operation is not guaranteed. |
| 28 | 6.3.2.3 | Pointer type conversion-function pointer | You cannot write a pointer to a function. |
| 29 | 6.4 | Lexical element | The character '' 'or the character' '' does not match the type of its last classification (delimiter, and lexically other preprocessed tokens  A message is output if it falls within a single non-blank character). |
| 30 | 6.4.1 | keyword | It cannot be specified except for keywords. I will error or ignore. |
| 31 | 6.4.2.1 | Identifier of out-of-range international character name | International character names are not supported. |
| 32 | 6.4.2.1 | International character name identifier whose first character represents a number | International character names are not supported. |
| 33 | 6.4.2.1 | Different identifiers for meaningless characters | There are no meaningless characters, because all the characters in the identifier are meaningful characters. |
| 34 | 6.4.2.2 | Predefined Identifier | \_\_func\_\_ cannot be described. It will be a compilation error. |
| 35 | 6.4.5 | String literals-change | String literals are not supported. |
| 36 | 6.4.7 | Header name | The characters ',', /, or / \* appear in the string between the delimiters <and> or in the string of the two delimiters “ In such cases, it is treated as a file name as it is. Treats the \ character as a folder separator and does not interpret it as an escape character. |
| 37 | 6.5 | Formula-Update multiple values ​​during evaluation | If the value of the object is changed twice or more while evaluating the expression, the operation is not guaranteed. |
| 38 | 6.5 | Formula-Exception occurred during evaluation | If an exception occurs during expression evaluation, it is undefined. Operation is not guaranteed. |
| 39 | 6.5 | Expression-access by lvalue with an unallowed type | If the value stored in the object is accessed by an lvalue with a disallowed type, the behavior is not guaranteed. |
| 40 | 6.5.2.2  6.5.15  6.5.16  6.5.17 | Formula-change of specific operation result | If you try to change the result of a function call, the result of a conditional operator, the result of an assignment operator, the result of a comma operator, or try to access after the next side effect completion point, an error occurs. |
| 41 | 6.5.2.2 | Function call-number of arguments | In the case where the calling function has a type that does not include the function prototype,  When the number of actual arguments is small, dummy arguments lacking corresponding actual arguments have undefined values.  If there are many actual arguments, the function is executed with no extra actual arguments, and the actual arguments have no meaning. |
| 42 | 6.5.2.2 | Function call-type of argument when calling function including function prototype | If a function is defined with a type including function prototype, and the type of the actual argument after expansion does not match the type of the dummy argument after expansion, it is interpreted as the type of dummy argument. |
| 43 | 6.5.2.2 | Function call-The type of argument when calling a function that does not contain the function prototype | Description that does not include function prototype (old form) is not supported. |
| 44 | 6.5.2.2 | Function call-incompatible function type | If a function is defined with a type incompatible with the type pointed to by the expression that represents the function to be called, the return value of the function will be invalid. |
| 45 | 6.5.3.2 | Address and Indirect Operator | If an incorrect value is assigned to a pointer, the behavior of the unary \* operator will output a message, take an undefined value, or cause an incorrect access depending on the hardware design and the content of the incorrect value. . |
| 46 | 6.5.4 | Cast from pointer | Conversion of type other than pointer type integer type or pointer type results in an error. |
| 47 | 6.5.5 | Divide by 2 the value of the second operand | A compile error will occur if the value of the second operand of the / operator is 0. |
| 48 | 6.5.5 | Remainder of which the value of the second operand is 0 | The% operator is not supported. |
| 49 | 6.5.6 | Additive operator-non-array pointer  Additive operator-operation result is outside the array | Even if the result of the pointer operation does not point to an element of the array and does not point beyond the last element of the array, it behaves as if it points to an element of the array. |
| 50 | 6.5.6 | Additive operator-a reference beyond the last element of the array | When the result of addition and subtraction on a pointer pointing to an element of an array object points to a point beyond the last element of the array, the result is evaluated by the unary \* operator Operation is not guaranteed. |
| 51 | 6.5.6 | Additive operator-pointer subtraction to another array | When subtraction is performed on two pointers other than the ones acting as if they point in the same array object, it behaves as if it were pointing to an element of the array. |
| 52 | 6.5.6 | Array subscript | Even if the index of the array is out of range, it behaves as if it points to an element of the array. If the subscript is a variable, no warning message is issued. |
| 53 | 6.5.6 | Pointer type subtraction type | If the result of subtraction of two pointers cannot be represented by an object of type ptrdiff\_t, ptrdiff\_t cannot correctly represent the value. |
| 54 | 6.5.7 | Bitwise shift operator-value of right operand | If the value of the right operand is negative or greater than or equal to the bit width of the expanded left operand, the shift result is undefined. |
| 55 | 6.5.7 | Bitwise shift operator-left operand value | When an expression with a signed extension type is shifted left, if the value of the expression is negative, the result is the shifted result. If the result of the shift cannot be represented by the extended type, the part that cannot be represented is truncated. |
| 56 | 6.5.8 | Relational operator-pointer | If the objects pointed to by the pointers to be compared are not members of the same aggregate object or union object, it operates as a relation operation between pointers pointing to the same object. |
| 57 | 6.5.16.1 | Simple substitution | If the values ​​stored in an object are accessed through some other object that somehow overlaps the object's storage, the overlap must match exactly. In addition, the two object types must be qualified or unqualified versions of the compatible type. For non-matching overlap assignments, the assignment destroys the original value of the assignment. The value after assignment is not guaranteed. |
| 58 | 6.6 | Integer constant expression-type | If the type of an integer constant expression is not an integer type, a compilation error will occur. |
| 59 | 6.6 | Integer constant expression-operand | A compile error will occur if the integer constant expression has any of the following operands:  (1) Integer constant  (2) Enumeration constant  (3) Character constant  (4) sizeof expression whose result is an integer constant  (5) Floating-point constant which is the direct operand of cast |
| 60 | 6.6 | Integer constant expression-cast operator | If the cast operator in an integer constant expression is not one that converts an arithmetic type to an integer type unless the cast operator is part of the sizeof operator's operand, a compile error will occur. |
| 61 | 6.6 | Form of constant expression in initializer | If the form of the constant expression in the initializer or the result of evaluating it is not any of the following, a compilation error will occur.  (1) Arithmetic constant expression  (2) Empty pointer constant  (3) Address constant  (4) Address constant for object type + integer constant expression  (5) Address constant for object type-integer constant expression |
| 62 | 6.6 | Arithmetic constant expression-type | If the arithmetic constant expression does not have an arithmetic type, a compilation error will occur. |
| 63 | 6.6 | Arithmetic constant expression-operand | If the arithmetic constant expression has any of the following operands, a compilation error will occur.  (1) Integer constant  (2) Floating point constant  (3) Enumeration constant  (4) Character constant  (5) sizeof formula |
| 64 | 6.6 | Arithmetic constant expression-Cast operator | If the cast operator in an arithmetic constant expression is not to convert an arithmetic type to an arithmetic type unless the result is part of the operand of the sizeof operator that is an integer constant, a compilation error will result. . |
| 65 | 6.6 | Address Constants-Value Access | If you access the value of an object by the following operators when generating address constants, the operation is not guaranteed.  (1) Array subscript operator "[]"  (2) Member access operator '.' And "->"  (3) Address unary operator '&'  (4) Indirect unary operator '\*'  (5) Cast to pointer |
| 66 | 6.7 | Declaration-Incomplete Type Declaration | If the object identifier is declared unbound and the object is of incomplete type, a compile error will occur. |
| 67 | 6.7.1 | Storage class specifier-declaration of function identifier | If a function identifier is declared in block scope with an explicit storage class specifier other than extern, a compile error will occur for auto / register. Static does not generate an error and is treated as a function declaration. |
| 68 | 6.7.2.1 | Type Specifier-Structure Specifier and Union Specifier | If the member declaration list does not contain a named member, a warning message will be output stating that it has no meaning.  Output a warning message that there is no named member. |
| 69 | 6.7.2.1 | Type specifier-flexible array member of a structure | When a flexible array member of a structure does not provide an element  (1) When the element of the array member is accessed, it operates as if it is pointing to the element.  (2) If you generate a pointer beyond that element, it operates as if it were pointing to that position. |
| 70 | 6.7.2.3 | Type specifier-incomplete type structure or union type | If a complete type is required, a compile-time error results if the incomplete type structure or union type is not complete without another declaration of the tag defining the content in the same scope. |
| 71 | 6.7.3 | Type modifier-const | If you try to change an object defined with const qualified type using an lvalue with non-const qualified type, a warning message will be output.   If you try to change an object defined with const qualified type using an lvalue of non-const qualified type, it may generate code to rewrite the object defined with const qualified type. The operation result is undefined. |
| 72 | 6.7.3 | Type modifier-volatile | If you try to modify an object defined with volatile qualified type using an lvalue of non-volatile qualified type, a warning message will be output.   If you try to modify an object defined with volatile qualified type using an lvalue of non-volatile qualified type, it may generate code to rewrite it as an lvalue of non-volatile qualified type. |
| 73 | 6.7.3 | Type modifier-function type | If you specify a function type that includes a type modifier, a warning message will be output stating that it has no meaning. |
| 74 | 6.7.3 | Type modifier-type match | If two qualified types that are required to conform are not the same modification to the conformed type, a warning is issued and acts as "do not conform". |
| 75 | 6.7.3.1 | Type modifier-restrict (access through pointer to const qualified type) | The behavior is not guaranteed if the modified object is accessed through a restricted qualified pointer to a const qualified type. |
| 76 | 6.7.3.1 | Type modifier-restrict (access through another pointer) | If the modified object is accessed through a restrict pointer and another pointer not based on the same object, the behavior is not guaranteed. |
| 77 | 6.7.3.1 | Constant expression-assignment between restrict qualified pointers | When assigning a pointer expression based on another restrict modified pointer (referred to as P2) to a restrict modified pointer (referred to as P), execution of the block (referred to as B2) bound to P2 is P The operation is not guaranteed if B2's execution does not complete before its assignment before it begins before the block bound to. |
| 78 | 6.7.4 | Function specifier-inline | Inline function specifiers cannot be written. Not supported |
| 79 | 6.7.5.1 | Pointer type-not compatible (type modifier) | If the two pointer types are not qualified identically, a warning message or an error message will be output. |
| 80 | 6.7.5.1 | Pointer type-nonconforming (type) | If the two pointer types are not pointers to compatible types, a warning message or an error message will be output. |
| 81 | 6.7.5.2 | Array declarator-An expression (number of elements) specifying the size of an array is not a constant expression | Variable length array type is not supported. |
| 82 | 6.7.5.2 | Array declarator-not compatible (element type) | In a context where two matching array types are required, if either element type does not match, a warning message or an error message will be output. Operation is not guaranteed. |
| 83 | 6.7.5.2 | Array declarator-not match (number of elements) | In contexts where two matching array types are required, if either size does not match, a warning or error message is output. Operation is not guaranteed. |
| 84 | 6.7.5.3 | Function declarator-dummy argument of array type | If the declaration of an array dummy argument includes the static keyword, and the value of the actual argument of the function does not point to the first element of the array having more than the number of elements, the incorrect value takes an undefined value Or, it will be illegal access. |
| 85 | 6.7.5.3 | Function declarator-modification of void of formal argument type list | Specifying a storage class specifier or type modifier in the dummy argument list as a void results in a compilation error. |
| 86 | 6.7.5.3 | Function declarator-return type | You cannot write a pointer to a function. |
| 87 | 6.7.5.3 | Function declarator-formal argument mismatch | You cannot write a pointer to a function. |
| 88 | 6.7.8 | Initializer-an unnamed member | Bit fields are not supported. |
| 89 | 6.7.8 | Initializer-scalar object | It is a compile error if the initializer for a scalar object is neither a single expression nor a single expression enclosed in braces. (A single expression enclosed by "(" and ")" does not cause an error) |
| 90 | 6.7.8 | Initializer-structure / union object with automatic storage period | If the initializer of a structure or union object with an automatic storage period is neither an initializer list nor a single expression of a compatible structure or union type, a compilation error or warning message is output. |
| 91 | 6.7.8 | Initializer-an object with an aggregate or union type | If the initializer for an object with an aggregate or union type is not the initializer list for the element or named member enclosed in braces, a compile error or warning message is output. |
| 92 | 6.9 | Identifier declared in outer join-no outer definition when used in an expression | If you refer to an identifier declared in an outer join, an error will occur when generating an executable file if there is no external definition. |
| 93 | 6.9 | Identifiers declared in outer join-multiple outer definitions when not used in an expression | Even if you do not refer to an identifier declared in an outer join, an error will occur at executable file generation if there are multiple external definitions. |
| 94 | 6.9.1 | Function definition-no declaration of formal argument type | If you do not specify a formal argument type, a compile error will occur. |
| 95 | 6.9.1 | Function definition-formal argument is a type other than object type | The result of typed dummy arguments will not be a non-object type (incomplete type or functional type). However, function types cannot be specified because they are typed into function pointers. |
| 96 | 6.9.1 | Function definition-variable arguments | You cannot write a function that takes a variable number of actual arguments. |
| 97 | 6.9.1 | Function definition-return statement | If it reaches "}" which terminates the function and the caller uses the value of the function call, an error will be output. |
| 98 | 6.9.2 | External object definition-an object with an internal connection | It is an error if the identifier for an incomplete type object with an inner join is tentatively defined. |
| 99 | 6.10.1 | Conditional capture | If the substitution processing generates a lexical defined, or if the usage before macro substitution of the unary operator does not match either of the two forms specified in the constraint, it is treated as a normal defined. |
| 100 | 6.10.2 | #include-#include pre-processing directive after macro replacement | If the #include preprocessing directive after macro replacement does not match either of the two header name formats (#include <string>, #include "string"), an error is output. |
| 101 | 6.10.2 | #include-start character in preprocessing directive | If the character string in the preprocessing command does not start with an alphabetic character, it is regarded as a file name as it is. |
| 102 | 6.10.3 | Macro replacement-actual arguments with preprocessing directives | If there is a preprocessing token string that acts as a preprocessing directive in other cases, a message will be output. |
| 103 | 6.10.3.2 | #operator | An error is output because the replacement result is a string. |
| 104 | 6.10.3.3 | ##operator | Outputs a message if the result of substitution does not result in a correct preprocessing token. |
| 105 | 6.10.4 | #line preprocessing directive-form violation | #line number string  #line number string "string"  Specifying #line in any format other than the above will result in an error. |
| 106 | 6.10.4 | #line pre-processing command-number specification outside the specified range | An error will occur if 0 is specified in the number string of #line.  If you specify a number greater than 4294967295 in the number string of #line, an error occurs. |
| 107 | 6.10.6 | Non-STDC pragma pre-processing command-operation not conforming to the standard | Prints compilation errors or warnings. |
| 108 | 6.10.6 | STDC pragma directive-form violation | Output a warning or an error.  Prescribed format:  # pragma STDC FP\_CONTRACT {ON | OFF | DEFAULT}  # pragma STDC FENV\_ACCESS {ON | OFF | DEFAULT}  # pragma STDC CX\_LIMITED\_RANGE {ON | OFF | DEFAULT} |
| 109 | 6.10.8 | Predefined macro name or defined identifier | An error is output when the predefined macro name or defined identifier is targeted for #define preprocessing directive or #undef preprocessing directive. In the case of a warning, the operation is not guaranteed. |
| 110 | clause 7 | Library function-copy between duplicate objects | In particular, library functions other than those permitted are not guaranteed to work if copied between duplicate objects. |
| 111 | 7.1.2 | Standard header-A file with the same name as a standard header | When a file with the same name as a standard header is placed at a file search position, an unsupported header name is imported. If it is a support header name, include the file at the early position of the path to search for include specification file. |
| 112 | 7.1.2 | Standard header-inclusion inside an external declaration or external definition | If the standard header is included inside an external declaration or external definition, it will be expanded as it is. |
| 113 | 7.1.2 | Standard header-Function / object / type / macro reference position to declare / define | If a function, object, type, or macro that the header declares or defines is referenced before the standard header is imported:  Treat as a normal undeclared identifier reference. |
| 114 | 7.1.2 | Standard header-Macro with the same name as the definition keyword | If you define a macro with the same name more than once, a warning will be issued to the second and subsequent macro definitions, and the latter definition will be valid. |
| 115 | 7.1.2 | Standard header-declaration of library function without outer binding | All declarations of library functions have external linkage.  ※ Same as [TBD] PEACE |
| 116 | 7.1.3 | Reserved Identifiers—Redeclaration / Redefinition of Reserved Identifiers | If you declare or define a reserved identifier, the behavior is not guaranteed. |
| 117 | 7.1.3 | Reserved Identifier-Invalidate macro definition with #undef | If the program invalidates the macro definition of the next identifier by #undef, an error will occur at the invalid macro usage location.  (1) Macro definition of identifier that starts with two underscores  (2) Macro definition of an identifier that begins with an uppercase letter following an underscore |
| 118 | 7.1.4 | Library Function Usage-Incorrect Actual Argument | If you call the library function with invalid actual arguments, the operation is not guaranteed. |
| 119 | 7.1.4 | Library function-a pointer passed to a dummy argument to be regarded as an array | The operation is not guaranteed if the pointer passed to the array dummy argument of the library function does not have a value that all address calculation and object access are actually valid. |
| 120 | 7.2 | assert macro-invalidating macro definition | If the assert macro definition is invalidated, the operation after the invalidated location is not guaranteed. |
| 121 | 7.2 | assert macro-actual argument | If the argument is not scalar type, an error is output. |
| 122 | 7.3.4  7.6.1  7.12.2 | CX\_LIMITED\_RANGE pragma  FENV\_ACCESS pragma  FP\_CONTRACT pragma | # pragma STDC CX\_LIMITED\_RANGE,  # pragma STDC FENV\_ACCESS  # pragma STDC FP\_CONTRACT  Will only check the grammar and ignore the content. |
| one two Three | 7.4 | Value of actual argument to character manipulation function | Character manipulation functions are not supported. |
| 124 | 7.5 | errno — disable macro definition | An error is output at errno usage point. |
| 125 | 7.5 | errno — redefine identifier | If redefined, the operation is not guaranteed. |
| 126 | 7.6.1 | FENV\_ACCESS Pragma-Floating Point Status Flag | Regardless of the state of the FENV\_ACCESS pragma,  The #pragma STDC FENV\_ACCESS specification is ignored. |
| 127 | 7.6.1 | FENV\_ACCESS pragma-floating point control mode | Regardless of the state of the FENV\_ACCESS pragma,  The #pragma STDC FENV\_ACCESS specification is ignored. |
| 128 | 7.6.1 | FENV\_ACCESS pragma-other than default state | Regardless of the state of the FENV\_ACCESS pragma,  The #pragma STDC FENV\_ACCESS specification is ignored. |
| 129 | 7.6.2 | Floating point exception-the value of excepts | Floating point exception functions are not supported. |
| 130 | 7.6.2.4 | fegetexceptflag-an unset fexcept\_t object | Floating point exception functions are not supported. |
| 131 | 7.6.4.3  7.6.4.4 | fesetenv / feupdateenv-unconfigured fenv\_t object | Floating point environment functions are not supported. |
| 132 | 7.8.2.1  7.8.2.2  7.8.2.3  7.8.2.4  7.20.6.1  7.20.6.2  7.20.1 | Integer arithmetic function or conversion function-a value whose result cannot be represented | If the value of the result of an integer arithmetic function or conversion function cannot be represented, an undefined value is returned. |
| 133 | 7.11.1.1 | The program changes the string pointed to by the pointer returned by the setlocale function | Culture specific functions are not supported. |
| 134 | 7.11.2.1 | The program changes the structure pointed to by the pointer returned by the localeconv function | Culture specific functions are not supported. |
| 135 | 7.12 | math\_errhandling — disable macro definition | Since generation of the floating point exception "inexact" is not supported, disabling it has no effect. |
| 136 | 7.12 | math\_errhandling — definition of math\_errhandling identifier | If math\_errhandling is defined as an identifier, a compile error will occur. |
| 137 | 7.12.3  7.12.14 | Actual argument of floating-point classification macro or comparison macro | When the actual argument is not a real floating point type, if the actual argument is a scalar type, it is converted to a float type and the corresponding macro is expanded. If it is not scalar type, an error will occur. |
| 138 | 7.13 | When setjmp macro definition is invalidated | Non-local branching functions are not supported. |
| 139 | 7.13 | When the external identifier setjmp is defined | Non-local branching functions are not supported. |
| 140 | 7.13.1.1 | Recovery of calling environment-calling setjmp macro at wrong place | Non-local branching functions are not supported. |
| 141 | 7.13.2.1 | Recovery of calling environment-Restoration of nonexistent environment | Non-local branching functions are not supported. |
| 142 | 7.13.2.1 | Calling environment-value change between setjmp and longjmp | Non-local branching functions are not supported. |
| 143 | 7.14.1.1 | The program passes an incorrect pointer to a signal processing routine function | Signal handling functions are not supported. |
| 144 | 7.14.1.1 | When the signal processing routine returns a signal corresponding to a calculation exception | Signal handling functions are not supported. |
| 145 | 7.14.1.1 | When a signal is generated as a result of calling abort function / raise function and the signal processing routine calls raise function | Signal handling functions are not supported. |
| 146 | 7.14.1.1 | When a signal occurs other than as a result of calling abort / raise function and the signal processing routine references an object with a static storage period  (Except when assigning a value to an object declared as volatile sig\_atomic\_t) | Signal handling functions are not supported. |
| 147 | 7.14.1.1 | When a signal is generated other than the result of calling abort / raise function, and the signal processing routine calls a function of the standard library  ( Except when the signal function itself is called with the signal number corresponding to the signal that generated the call to the abort function, \_Exit function, or signal processing routine as the first actual argument) | Signal handling functions are not supported. |
| 148 | 7.14.1.1 | When the value of errno is referenced in the case where the corresponding signal processing routine calls the signal function and the return value is SIG\_ERR after the signal other than the result of the abort function / raise function is generated | Signal handling functions are not supported. |
| 149 | 7.14.1.1 | When a signal is generated by an asynchronous signal handling routine | Signal handling functions are not supported. |
| 150 | 7.15  7.15.1.1  7.15.1.4 | Variable number of actual arguments-access to va\_list | Macros that handle variable arguments are not supported. |
| 151 | 7.15 | Variable number of actual arguments-access with separate functions | Macros that handle variable arguments are not supported. |
| 152 | 7.15.1 | Variable number of actual arguments-invalidation of macro definition | Macros that handle variable arguments are not supported. |
| 153 | 7.15.1 | Variable number of actual arguments-external identifier declaration with same name as macro | Macros that handle variable arguments are not supported. |
| 154 | 7.15.1  7.15.1.2  7.15.1.3  7.15.1.4 | Variable number of actual arguments-va\_end macro call | Macros that handle variable arguments are not supported. |
| 155 | 7.15.1.1 | Variable number of actual arguments-second argument of va\_arg macro | Macros that handle variable arguments are not supported. |
| 156 | 7.15.1.1 | Variable number of actual arguments-va\_arg macro (there is no next actual argument or the type of the next actual argument does not match) | Macros that handle variable arguments are not supported. |
| 157 | 7.15.1.2  7.15.1.4 | Variable number of actual arguments- call va\_copy or va\_start macro for reinitialization | Macros that handle variable arguments are not supported. |
| 158 | 7.15.1.4 | Variable number of actual arguments-type of second argument of va\_start macro | Macros that handle variable arguments are not supported. |
| 159 | 7.17 | offsetof macro-invalid member indicator | It is an error if the offsetof macro member directive is invalid. |
| 160 | 7.18.4 | Integer constant macro-over type limit | Expand the specified value as it is. |
| 161 | 7.18.4 | Integer constant macro-specified value of actual argument | Expand the specified value as it is. If you specify a value with a suffix, the suffix is ​​expanded as it is. However, the following macro expands as follows.  INT32\_C (value): ((value) + (INT32\_MAX-INT32\_MAX))  UINT32\_C (value): ((value) + (UINT32\_MAX-UINT32\_MAX)) |
| 162 | 7.19.2 | Byte I / O Function-Applies to a wide character stream | Because library functions that handle wide characters are not supported, they cannot be streams in wide character units. |
| 163 | 7.19.2 | Wide character input / output functions-apply to byte-wise streams | Library functions that handle wide characters are not supported. |
| 164 | 7.19.2 | Wide character stream-Use the portion of the file that exceeds the wide character Use from the position of the second byte of the wide character | Because library functions that handle wide characters are not supported, they cannot be streams in wide character units. |
| 165 | 7.19.3 | File-use of FILE after close | fclose function is not supported. |
| 166 | 7.19.5.2 | fflush function-if the output stream or the previous operation points to something other than an update stream that is not input | The fflush function is not supported. |
| 167 | 7.19.5.3 | fopen function-character with invalid mode specification | The fopen function is not supported. |
| 168 | 7.19.5.3 | fopen function-in update mode, if you do not flush the stream in the case of output-> input | The fopen function is not supported. |
| 169 | 7.19.5.3 | fopen function-In update mode, in the case of input-> output, when the file positioning function is not called | The fopen function is not supported. |
| 170 | 7.19.5.6 | setvbuf function-if the contents of the buffer array used in the call are used | The setvbuf function is not supported. |
| 171 | 7.19.6.1  7.19.6.2  7.24.2.1  7.24.2.2 | Formatted I / O Function-Insufficient Actual Argument to Format | Formatted I / O functions are not supported. |
| 172 | 7.19.6.1  7.19.6.2  7.24.2.1  7.24.2.2 | Formatted I / O function-invalid conversion specification | Formatted I / O functions are not supported. |
| 173 | 7.19.6.1  7.19.6.2  7.24.2.1  7.24.2.2 | Formatted I / O function-invalid for the conversion specification corresponding to the actual argument type | Formatted I / O functions are not supported. |
| 174 | 7.19.6.1  7.19.6.2  7.23.3.5  7.24.2.1  7.24.2.2  7.24.5.1 | Formatted I / O and strftime functions, wcsftime functions-not a sequence of multibyte characters whose format starts with an initial shift and ends with an initial shift | Formatted I / O functions are not supported. |
| 175 | 7.19.6.1  7.24.2.1 | Formatted output function-specify precision other than the specified conversion specifier | Formatted output functions are not supported. |
| 176 | 7.19.6.1  7.24.2.1 | Formatted output function-use of "\*" for field width or precision | Formatted output functions are not supported. |
| 177 | 7.19.6.1  7.24.2.1 | Formatted Output Function-Use of the # or 0 Flags Outside of the Default Conversion Specifier | Formatted output functions are not supported. |
| 178 | 7.19.6.1  7.19.6.2  7.24.2.1  7.24.2.2 | Formatted I / O Function-Specifying a Length Modifier Other Than the Specified Conversion Specifier | Formatted I / O functions are not supported. |
| 179 | 7.19.6.1  7.24.2.1 | Formatted output function-corresponding actual argument at s conversion specifier | Formatted output functions are not supported. |
| 180 | 7.19.6.1  7.19.6.2  7.24.2.1  7.24.2.2 | Formatted input / output function-n conversion specification | Formatted I / O functions are not supported. |
| 181 | 7.19.6.1  7.19.6.2  7.24.2.1  7.24.2.2 | Formatted input / output function-% conversion specification | Formatted I / O functions are not supported. |
| 182 | 7.19.6.1  7.19.6.2  7.23.3.5  7.24.2.1  7.24.2.2  7.24.5.1 | Formatted I / O function / strftime function / wcsftime function-invalid conversion specification | Formatted I / O functions are not supported. |
| 183 | 7.19.6.1  7.19.6.3  7.19.6.8  7.19.6.10 | Formatted Output Function-Number of Output Characters | Formatted output functions are not supported. |
| 184 | 7.19.6.2  7.24.2.2 | Formatted Input Function-Unrepresentable Conversion Result | Formatted input functions are not supported. |
| 185 | 7.19.6.2  7.24.2.2 | Formatted Input Function-Receiving Incorrect Conversion Results | Formatted input functions are not supported. |
| 186 | 7.19.6.2  7.24.2.2 | Formatted Input Function-c, s, [conversion specifier | Formatted input functions are not supported. |
| 187 | 7.19.6.2  7.24.2.2 | Formatted input function-if the input is not a valid multibyte character sequence | Formatted input functions are not supported. |
| 188 | 7.19.6.2  7.24.2.2 | Formatted input function-% p conversion | Formatted input functions are not supported. |
| 189 | 7.19.6.8  7.19.6.9  7.19.6.10  7.19.6.11  7.19.6.12  7.19.6.13  7.19.6.14  7.24.2.5  7.24.2.6  7.24.2.7  7.24.2.8  7.24.2.9  7.24.2.10 | Formatted input / output function (va\_list argument)-use of improperly initialized va\_list actual argument | The functions vfprintf, vfscanf, vprintf, vscanf, vsnprintf, vsprintf, vsscanf, vfwprintf, vfwscanf, vswprintf, vswscanf, vwprintf, and vwscanf are not supported. |
| 190 | 7.19.6.8  7.19.6.9  7.19.6.10  7.19.6.11  7.19.6.12  7.19.6.13  7.19.6.14  7.24.2.5  7.24.2.6  7.24.2.7  7.24.2.8  7.24.2.9  7.24.2.10 | Formatted I / O function (va\_list argument)-using actual argument without calling va\_end | The functions vfprintf, vfscanf, vprintf, vscanf, vsnprintf, vsprintf, vsscanf, vfwprintf, vfwscanf, vswprintf, vswscanf, vwprintf, and vwscanf are not supported. |
| 191 | 7.19.7.2  7.19.7.7  7.24.3.2 | String Input Function-Using Input Content After a Read Error Occurs | fgets, gets, fgetws functions are not supported. |
| 192 | 7.19.7.11 | ungetc function-call when the value of file position indicator is 0 | The ungetc function is not supported. |
| 193 | 7.19.8.1  7.19.8.2 | fread function / fwrite function-when file position indicator is used after an error occurs | fread function and fwrite function are not supported. |
| 194 | 7.19.8.1 | fread function-when reading and using only part of one element | The fread function is not supported. |
| 195 | 7.19.9.2 | fseek function-when text stream, offset is not the return value of ftell function | fseek function is not supported. |
| 196 | 7.19.9.2 | fseek function-when text stream, when when not SEEK\_SET | fseek function is not supported. |
| 197 | 7.19.9.3 | fgetpos function-if the value of pos is not the return value of fgetpos function | The fgetpos function is not supported. |
| 198 | 7.20.3 | Dynamic memory management function-use pointer obtained with request size 0 | Dynamic memory management functions are not supported. |
| 199 | 7.20.3 | Dynamic Memory Management Functions-Referencing Freed Space | Dynamic memory management functions are not supported. |
| 200 | 7.20.3.2  7.20.3.4 | Dynamic Memory Management Functions-Mismatched Release and Get Pointers | Dynamic memory management functions are not supported. |
| 201 | 7.20.3.2  7.20.3.4 | Dynamic Memory Management Function-Release Freed Space | Dynamic memory management functions are not supported. |
| 202 | 7.20.3.3 | Dynamic memory management function-reference to malloc function allocation area | Dynamic memory management functions are not supported. |
| 203 | 7.20.3.4 | Dynamic memory functions-reference new objects allocated by the realloc function | Dynamic memory management functions are not supported. |
| 204 | 7.20.4.3 | exit function-multiple calls | It will not be called more than once. |
| 205 | 7.20.4.3 | atexit function-longjmp function call | The atexit function is not supported. |
| 206 | 7.20.4.5  7.21.6.2 | getenv function / strerror function-change the returned error message | Both getenv and strerror functions are not supported. |
| 207 | 7.20.4.6 | system function-execute system function with exit or undefined behavior | The system function is not supported. |
| 208 | 7.20.5 | bsearch function / qsort function-element is 0 and illegal pointer call | Both bsearch and qsort functions are not supported. |
| 209 | 7.20.5 | bsearch function / qsort function-change the contents of the search / alignment target array with the comparison function | Both bsearch and qsort functions are not supported. |
| 210 | 7.20.5 | bsearch function / qsort function-result of comparison function is incorrect | Both bsearch and qsort functions are not supported. |
| 211 | 7.20.5.1 | bsearch function-the order of the elements of the searched array is incorrect | The bsearch function is not supported. |
| 212 | 7.20.7 | Multi-byte character / wide character conversion function-use in current conversion state after changing LC\_CTYPE category | LC\_CTYPE is not supported. |
| 213 | 7.21.1  7.24.4 | String / Wide String Utility Functions-Array Access | It is described in "Library Function". |
| 214 | 7.21.1  7.24.4 | String / Wide String Utility Function-Incorrect pointer call with value 0 of size\_t type | It is described in "Library Function". |
| 215 | 7.21.4.5  7.23.3.5  7.24.4.4.4  7.24.5.1 | strxfrm function / strftime function / wcsxfrm function / wcsftime function-use the contents of copy destination array with insufficient length | The strxfrm function, strftime function, wcsxfrm function, and wcsftime function are not supported. |
| 216 | 7.21.5.8  7.24.4.5.7 | strtok function / wcstok function-the first actual argument on the first call is an empty pointer | The strtok function and wcstok function are not supported. |
| 217 | 7.22 | Type generic macro-if the type of the actual argument does not match the type of the dummy argument | Type-generic mathematical functions are not supported. |
| 218 | 7.22 | Type generic macro-when a generic formal argument is given a real argument of complex number | Type-generic mathematical functions are not supported. |
| 219 | 7.24.2.11 | fwprintf function-s specifier without l modifier | The fwprintf / wprintf functions are not supported. |
| 220 | 7.24.4.5.7 | wcstok function-the object pointed to by the third argument | The wcstok function is not supported. |
| 221 | 7.24.6 | mbstate\_t object | mbstate\_t type is not supported. |
| 222 | 7.25.1 | Value of actual argument of type wint\_t | The wint\_t type is not supported. |
| 223 | 7.25.2.2.1 | iswctype function-call with different LC\_CTYPE category | The iswctype function is not supported. |
| 224 | 7.25.3.2.1 | towctrans function-call with different LC\_CTYPE category | The towctrans function is not supported. |

## Processing system-dependent behavior

|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Standard chapter number** | **Item** | **Specification** |
| 1 | 3.10  5.1.1.3 | Diagnostic message output method | Outputs an error message including the source file name and line number (if it can be specified) for a translation unit that contains any syntax rule violation and constraint violation. The error message format is divided into "warning", "fatal error", "error" and so on. |
| 2 | 5.1.1.2 | A sequence of whitespace characters excluding newline characters | “A non-empty sequence of non-new-space characters other than newline characters” is not replaced by one but is retained as it is. |
| 3 | 5.1.1.2 | How to map multibyte characters of physical source file and corresponding source character set | Multi-byte characters are mapped to the corresponding source character set by the compiler option. |
| 4 | 5.1.2.1 | Name and type of function called at program start | The function (entry function) to be called first when the program starts can be specified with #pragma entry. |
| 5 | 5.1.2.1 | Effect of program termination processing | The abort function, exit function, and atexit function are pending because their specifications are undecided.   Use the abort function to terminate the program abnormally.  Use the exit function to terminate the program normally.  The atexit function is not supported. |
| 6 | 5.1.2.2.1 | Host environment-how to define the main function | CCIMP is not applicable due to the free standing environment. |
| 7 | 5.1.2.2.1 | Host environment-String value pointed to by argv argument of main function | CCIMP is not applicable due to the free standing environment. |
| 8 | 5.1.2.3 | Configuration of interactive device | The configuration of the interactive device is not specified.  Therefore, it depends on user-own coding or target system. |
| 9 | 7.14 | A set of entire signals, their meanings and default operations | The signal function is not supported. |
| 10 | 7.14.1.1 | Value of signal corresponding to calculation exception | The signal function is not supported. |
| 11 | 7.14.1.1 | Signal executed at program start | The signal function is not supported. |
| 12 | 7.20.4.5 | getenv function-how to change the set of defined names of environments and the list of environments | The getenv function is not supported. |
| 13 | 7.20.4.6 | system function-how to execute the string pointed to by the argument string | The system function is not supported. |
| 14 | 6.4.2 | Multibyte character identifier | Multibyte characters cannot be used as identifiers. |
| 15 | 5.2.4.1  6.4.2 | Identifier length | There is no limit on the length of the identifier. |
| 16 | 3.6 | Number of bits per byte | The number of bits per byte is 8 bits. |
| 17 | 5.2.1 | Character set element value | The values ​​of elements in the source character set and execution character set are ASCII code, EUC, SJIS, UTF-8, big5, gb2312.  Supports Japanese / Chinese description in comments and strings. |
| 18 | 5.2.2 | Unique value of an element of the execution character set assigned to alphabetic reverse slash notation | The extended notation values ​​are defined as follows.  Extended notation: Value (ASCII): Meaning  "\ a": 07: Alert (warning sound)  "\ b": 08: Backspace  "\ f": 0C: Form feed (Page feed)  "\ n": 0A: New line (line feed)  "\ r": 0D: Carriage return (return)  "\ t": 09: Horizontal tab  "\ v": 0B: vertical tab |
| 19 | 6.2.5 | Value of object of char type storing character other than any element of execution basic character set | It is a value converted to char type. A simple char type without a type specifier (signed, unsigned) has the same range of values ​​as a signed integer (signed char). |
| 20 | 6.2.5  6.3.1.1 | presence or absence of char type code | There is a sign. The char type has the same value as the signed char type. |
| 21 | 6.4.4.4  5.1.1.2 | How to map elements of source character set in character constant and string literal to elements of execution character set | Elements of the source character set of the character constant and elements of the character set in the execution environment match. However, you cannot write string literals. |
| 22 | 6.4.4.4 | The value of a simple character constant that contains multiple characters, or that cannot be represented by a single-byte execution character, or includes reverse slash notation | Simple character constants containing up to four characters have an int value with the last character in the low byte and the first character in the high byte. A character constant with five or more characters results in an error. An octal escape sequence if it contains characters and escape sequences that are not represented in the basic execution environment character set, and a hexadecimal escape sequence are the values ​​shown in its octal notation and hexadecimal notation. |
| 23 | 6.4.4.4 | The value of a wide character constant that contains two or more multibyte characters, or that cannot be represented in the execution extended character set, or that contains multibyte characters or reverse slash notation | The value is the leftmost single character (not byte) value. |
| 24 | 6.4.4.4 | Locale at which point the value of a wide character constant containing single multibyte characters corresponding to one character of the execution extended character set corresponds to the wide character | Locale (locale) is not supported. |
| 25 | 6.4.5 | Current locale used when converting from a wide string literal to the corresponding wide character code | Locale (locale) is not supported.  String literals are not supported. |
| 26 | 6.4.5 | Value of a string literal that contains multibyte characters or reverse slash characters that cannot be represented by the execution character set | String literals are not supported. |
| 27 | 6.2.5 | Extended signed integer type provided by the processing system | Extended signed integer types are not provided. |
| 28 | 6.2.6.2 | Representation method of signed integer type and its meaning | Signed integer types are represented by 2's complement. In this case, the number whose sign bit is 1 and all the value bits are 0 is a normal value. |
| 29 | 6.3.1.1 | Order between extended signed integer types with equal precision | Extended signed integer types are not provided. |
| 30 | 6.3.1.3 | When converting an integer type value to a signed integer type, when the value cannot be represented by a signed integer type | It is assumed that the bit string is masked (the upper bits are truncated) by the width of the number of small size bits. |
| 31 | 6.5 | Results of bitwise operations on signed integers | A shift right operation on a signed integer type performs an arithmetic shift. |
| 32 | 5.2.4.2.2 | Accuracy of library functions that return floating point operations and floating point results | Accuracy is unknown. |
| 33 | 5.2.4.2.2 | FLT\_ROUNDS-rounding behavior characterized by non-standard values | FLT\_ROUNDS does not define non-standard values. |
| 34 | 5.2.4.2.2 | FLT\_EVAL\_METHOD-an evaluation form characterized by non-standard negative values | FLT\_EVAL\_METHOD does not define non-standard values. |
| 35 | 6.3.1.4 | Floating point and integer | When converting an integer to a floating-point type that cannot be accurately represented to the original value, either nearest round (even selection) or 0-direction rounding can be selected. |
| 36 | 6.3.1.5 | double and float | The double type is treated as the same size as the float type. |
| 37 | 6.4.4.2 | Result when the floating-point constant is decimal floating-point constant or when hexadecimal FLT\_RADIX is not a power of 2 | For decimal floating-point constants, the result is rounded according to the specification of the -round option.  It does not specify the result of hexadecimal floating point determination when FLT\_RADIX is not a power of two.  Since FLT\_RADIX is a power of 2, it is assumed to be exactly rounded. |
| 38 | 6.5 | FP\_CONTRACT pragma-Floating point reduction | The FP\_CONTRACT pragma does not work.  Even if #pragma STDC FP\_CONTRACT is specified, it is ignored. |
| 39 | 7.6.1 | FENV\_ACCESS pragma-default state | The default state of the FENV\_ACCESS pragma is ON.  However, even if #pragma STDC FENV\_ACCESS is specified, it is ignored. |
| 40 | 7.6  7.12 | Add floating point exceptions, rounding mode, environment, classification, and their macro names | There is no floating point exception, rounding mode, environment, classification, and the addition of their macro names. |
| 41 | 7.12.2 | FP\_CONTRACT pragma-default state | The default state of the FP\_CONTRACT pragma is ON.  However, #pragma STDC FP\_CONTRACT specification is ignored. |
| 42 | F.9 | Generate an "incorrect result" floating-point exception when the rounding result is actually equivalent to the mathematical result | Floating point exceptions are not generated. |
| 43 | F.9 | Generate "underflow" floating point exceptions and "incorrect results" floating point exceptions when the result is very small but not incorrect | Floating point exceptions are not generated. |
| 44 | 6.3.2.3 | The result of an integer to pointer, pointer to integer conversion | (1) Result of conversion from integer to pointer  Integer type size == pointer type size  The integer bit pattern is kept as it is.  Integer type size <pointer type size  The value of the result of extending an integer type value to an int type is retained as it is.  (2) Result of conversion from pointer to integer  Pointer type size> Integer type size  It is the value of the low byte of pointer type.  Pointer type size == integer type size  The pointer type bit pattern is kept as it is. |
| 45 | 6.5.6 | Resulting size of subtraction of two pointers to elements of the same array | The result is of type signed int. |
| 46 | 6.7.1 | Usage effect of the register storage class specifier | Optimization is performed to access as fast as possible regardless of the storage class specifier "register" declaration. |
| 47 | 6.7.4 | Usage effect of inline function specifier | Inline function specifiers are not supported. |
| 48 | 6.7.2  6.7.2.1 | Handling of just int bit field | Bit fields are not supported. |
| 49 | 6.7.2.1 | Type specifiers allowed in bit fields | Bit fields are not supported. |
| 50 | 6.7.2.1 | Space allocation in the case of exceeding boundaries that satisfy storage units in bit field declaration | Bit fields are not supported. |
| 51 | 6.7.2.1 | Order of bit field allocation in unit | Bit fields are not supported. |
| 52 | 6.7.2.1 | How to adjust the boundaries of each member of a structure or union object except bit fields | char is 1, short is 2, int and long will be adjusted to the boundary of 4 bytes. |
| 53 | 6.7.2.2 | An integer that represents an enumerated value | It is assumed to be int type. |
| 54 | 6.7.3 | Access to volatile qualified objects | It depends on the address to which the data is mapped. There is no optimization for access to the same variable.  Access width, access order and access count are implemented as described in C source. |
| 55 | 6.4.7 | How is the string of header name specified by #include is mapped to the name of header or external source file | The character string described in the #include header is a character that can be used in the source character set, and is mapped to the header name or the external source file name. |
| 56 | 6.10.1 | Values ​​for character constants specified by conditional embedding and values ​​of character constants appearing in other expressions | The value for the character constant specified in the condition embedding and the value of the character constant appearing in other expressions are equal. |
| 57 | 6.10.1 | Character constant value specified by conditional embedding | If a simple char type (signed or unsigned char or not) is unsigned, it has no negative value. If it is signed, it may have a negative value. |
| 58 | 6.10.2 | Preprocessing instruction of the form "#include <string>" | Search in the following order:  (1) In the case of full path specification, the folder indicated by that path  In the following order if there is no full path specification  (2) Folder specified in the include option  (3) Standard include file folder |
| 59 | 6.10.2 | Preprocessing instruction in the form of "#include" string "" | Search in the following order:  (1) In the case of full path specification, the folder indicated by that path  In the following order if there is no full path specification  (2) Folder with source file  (3) Folder specified in the include option  (4) Standard include file folder |
| 60 | 6.10.2 | "#Include preprocessed token string" format | It is treated as a preprocessing header for a single header file name only if the preprocessing token string is a single macro that is replaced with <string> or "string" format. |
| 61 | 6.10.2 | Restrictions on nesting of #include directives | There is no limit. |
| 62 | 6.10.3.2 | Correspondence of international character name in character constant and string literal of # operator | International character names are not supported. |
| 63 | 6.10.6 | Non-STDC pragma directive behavior | Please refer to "# pragma directive". |
| 64 | 6.10.8 | Date and time obtained by \_\_DATE\_\_ and \_\_TIME\_\_ macros when translation date and time cannot be obtained | The \_\_DATE\_\_ and \_\_TIME\_\_ macros are not supported. |
| 65 | 5.1.2.1 | Library functions that programs can use in a free standing environment | See "Library Functions". |
| 66 | 7.2.1.1 | Format of assert macro | The format is as follows. c89 and c99 specifications are not compliant.  ASSERTION FAILED: LINE LINE <<line number> |
| 67 | 7.6.2.2 | Representation of floating point status flags stored by the fegetexceptflag function | The fegetexceptflag function is not supported. |
| 68 | 7.6.2.3 | Behavior of floating-point exception of feraiseexcept function | The feraiseexcept function is not supported. |
| 69 | 7.11.1.1 | When passing a string other than "C" or "" in the string passed as the second argument to the setlocale function | The setlocale function is not supported. |
| 70 | 7.12 | Types defined by float\_t and double\_t when the value of the FLT\_EVAL\_METHOD macro is less than 0 or greater than 2 | float\_t is float type. double\_t is undefined. |
| 71 | 7.12.1 | Domain errors other than those required by the international standard (C99) in mathematical functions | L gamma f corresponds. |
| 72 | 7.12.1 | Return value of mathematical function when domain error occurs | lgamma f / tgamma f returns HUGE\_VAL, HUGE\_VALF, HUGE\_VALL with a sign.  Otherwise it returns a non-number. |
| 73 | 7.12.1 | Behavior of mathematical functions when the result of floating point operation underflows | The return value is 0.  When underflow occurs, errno is set to ERANGE.  Underflow floating-point exceptions are not generated. |
| 74 | 7.12.10.1 | fmod function group-operation when the second actual argument is 0 | A domain error occurs. The return value is not a number. EDOM is set to global variable error. |
| 75 | 7.12.10.3 | remquo function group-logarithm based on 2 | It is 31. |
| 76 | 7.14.1.1 | Operation at the time of signal generation | The signal function is not supported. |
| 77 | 7.17 | Empty pointer after macro NULL expansion | It will be 0. |
| 78 | 7.19.2 | Text stream-the terminating newline character on the last line | File manipulation functions are not supported. |
| 79 | 7.19.2 | Text stream-a sequence of whitespace characters written immediately before a newline character | File manipulation functions are not supported. |
| 80 | 7.19.2 | Binary stream-number of null characters added at the end | File manipulation functions are not supported. |
| 81 | 7.19.3 | File position indicator position for files opened in append mode | File manipulation functions are not supported. |
| 82 | 7.19.3 | Text stream-handling of final writing point in writing | File manipulation functions are not supported. |
| 83 | 7.19.3 | File buffering characteristics | File manipulation functions are not supported. |
| 84 | 7.19.3 | Existence of zero-length files | File manipulation functions are not supported. |
| 85 | 7.19.3 | Correct file name rules | File manipulation functions are not supported. |
| 86 | 7.19.3 | Simultaneous opening of the same file | File manipulation functions are not supported. |
| 87 | 7.19.3 | Representation characteristics and selection used for multibyte characters in the file | Shift state is not supported as a representation format of multi-byte characters. |
| 88 | 7.19.4.1 | remove function-operation on an open file | The remove function is not supported. |
| 89 | 7.19.4.2 | rename function-operation when the specified file already exists | The rename function is not supported. |
| 90 | 7.19.4.3 | Delete temporary files if program terminates abnormally | The tmpfile function is not supported. |
| 91 | 7.19.5.4 | freopen function-operation when the file name is an empty pointer | The freopen function is not supported. |
| 92 | 7.19.6.1, 7.24.2.1 | Formatted output-conversion format when writing infinity and NaN, and the meaning of n string and n wide string used in writing NaN | Formatted I / O functions are not supported. |
| 93 | 7.19.6.1  7.24.2.1 | fprintf function / fwprintf function-Output of% p conversion | Formatted I / O functions are not supported. |
| 94 | 7.19.6.2  7.24.2.1 | fscanf function / fwscanf function-interpretation of character-(hyphen) in "% [" conversion | Formatted I / O functions are not supported. |
| 95 | 7.19.6.2  7.24.2.2 | fscanf function / fwscanf function-set of matching character strings by "% p" conversion and interpretation of corresponding input items | Formatted I / O functions are not supported. |
| 96 | 7.19.9.1  7.19.9.3  7.19.9.4 | fgetpos function / fsetpos function / ftell function-value of errno macro set on failure (positive value) | The fgetpos function, fsetpos function and ftell function are not supported. |
| 97 | 7.20.1.3  7.24.4.1.1 | strtod function group-meaning of n character or n wide character string in character string representing converted NaN | The functions strtod, strtof, strtold, wcstod, wcstof and wcstold are not supported. |
| 98 | 7.20.1.3  7.24.4.1.1 | strtod functions-whether ERANGE is stored in errno when underflow occurs | The functions strtod, strtof, strtold, wcstod, wcstof and wcstold are not supported. |
| 99 | 7.20.3 | Dynamic memory management function-return value when the size of the area requested by the memory allocation function is 0 | Dynamic memory management functions are not supported. |
| 100 | 7.20.4.1  7.20.4.4 | abort function / \_Exit function-operation on open file and temporary file | abort , \_Exit does not flush the file stream. It also does not close temporary files. |
| 101 | 7.20.4.1  7.20.4.3  7.20.4.4 | abort function / exit function / \_Exit function-Exit status returned to host environment | The exit function, abort and \_Exit function do not return values. |
| 102 | 7.20.4.6 | system function-the value returned by the system function if the actual argument passed is not a null pointer | The system function is not supported. |
| 103 | 7.23.1 | Local time and summer time | time.h is not supported. |
| 104 | 7.23 | clock\_t / time\_t-representable time range and precision | time.h is not supported. |
| 105 | 7.23.2.1 | clock function-start of calculated processor time | time.h is not supported. |
| 106 | 7.23.3.5  7.24.5.1 | strftime function / wcsftime function-replacement string corresponding to% Z conversion specifier in "C" locale | time.h is not supported. |
| 107 | F.9 | Trigonometric function / hyperbolic function / exponential function with base e / logarithmic function with base e / error function / log gamma function-"incorrect result" floating point exception generation | Does not generate "inexact result" floating point exceptions. |
| 108 | F.9 | Rounding direction of functions in <math.h> | Follow rounding direction mode.   The function fesetround () that changes the rounding mode during program execution is not supported. It follows the compile-time rounding options. |
| 110 | 6.2.6.1 | The number of bytes in the sequence of bytes that make up the object, if not explicitly specified, the byte order, the representation method | The internal representation of the data is explicit. There is nothing that is not explicitly stated. |

# Functional Specifications

## Introduction

### About this Document

This document describes system specifications of V3.00.00 of the compiler for the IMP-X5+ shader core (hereafter referred to as ccimp).

The contents defined in this document are as follows:

* Specifications related to the entire compiler system (including the assembler and subsequent phases)
* Detailed specifications (C language specifications are however excluded) of the compiler phase (from the preprocessor to the code generator)

Details of the C language specifications accepted by the ccimp and specifications of the assembler and subsequent phases are defined in other documents.

The descriptions in this document are regardless of version unless otherwise stated. The specifications which differ with each version are clearly stated in this document.

## About the Product

### Position of Product

The ccimp is a C compiler package for the IMP-X5+ shader core.

As one module of the integrated development environment (IDE) for ADAS from Renesas Electronics Corporation (e2 studio for ADAS), the ccimp operates closely in cooperation with other modules (debugger, etc.) of e2 studio for ADAS.

In this case, the ccimp command is started via the IDE of e2 studio for ADAS.

Note that the ccimp command can also be started from the command line as an independent command.

### Feature

The ccimp is a high-performance compiler, conforming to the ANSI-C standard, in which object performance and compilation time are well balanced. The main features are as follows:

* High-performance

Global optimization is achieved in addition to conventional optimization by applying the new compiler architecture platform PEACE (Platform and Engine for Advanced Compiler Engineering) that was developed based on LLVMNote. This allows generation of code with a small code size and fast execution speed and also reduces the compilation time.

Note Low Level Virtual Machine: Free compiler infrastructure being developed at the University of Illinois in the U.S.

* Multifunctional

Unique instructions of the IMP-X5+ shader core can be used in built-in functions.

### Target Device

The ccimp supports the IMP-X5+ shader core which is an image-recognition IP equipped in R-Car V3M SoC.

### Supported Environment

* Hardware environment

Conforms to the operating environment of e2 studio for ADAS.

* Operating system

Conforms to the operating environment of e2 studio for ADAS.

### System Configuration

(Not open to users)

Figure 2.1 shows the configuration of the ccimp.

The ccimp command controls the phases from the compiler to the assembler or linker.

ccimp command

Front end

(rcfrt.exe)

Preprocessor

Syntax analyzer

Bitcode converter

CCRL driver

(ccimp.exe)

Assembler

(sasm.exe)

Linker

(slink.exe)

PEACE

(rcbackimp.exe)

Machine-  
independent phase

Machine-  
dependent phase

Code generator

Figure 2.1 System Configuration

### Package Configuration

The configuration of the ccimp package is as follows:

* ccimp command

Used to control the compiler, assembler, and linker.

* Compiler

Consists of the front end (rcfrt.exe) and back end (rcbackimp.exe).

* Assembler

Consists of only the assembler (sasm.exe). The assembler can be started without the IDE or compiler.

* Linker

Consists of only the linker (slink.exe). The linker can be started without the IDE or compiler.

* Attached libraries (including the header file)
* Standard libraries

Library file containing the following functions

* + C standard library for v3m2 (including single precision floating point math function library ) ( libc\_v3m2 )
  + C standard library for v3h (including single precision floating point math function library ) ( libc\_v3h )
  + C standard library for v3u (including single precision floating point math function library ) ( libc\_v3u)

## Outline of Processing

### Processing Flow

The ccimp uses source files written in the C language or assembly language to generate files executable on the target system.

In the ccimp, a single driver controls all phases; from the front end to the linker.

The processing flow in the ccimp is as shown below. However, the actual flow may differ from that shown below depending on the options being used at the start of the command.

1. The front end processes the comments and preprocessing directives in a C source file.

2. The front end converts the preprocessed C source file into internal data representations (bitcode: Not open to users) for the back end (PEACE).

3. The machine-independent phase, machine-dependent phase, and code generator are executed according to the optimization options. The code generator converts the internal data representations into an assembly source file suitable for each target.

4. The assembler converts the assembly source file into machine instructions and generates a relocatable object file. An assembly list file is also output at this time, if the relevant option is specified.

5. The linker links the object files and generates a load module file executable in each target.

Figure 2.2 shows the processing flow.

.c

.s

Assembler

Linker

IDE

Debugger

.o

.out

Other modules of e2 studio

Preprocessor

Compiler phase

Should be prepared by the user

Simulator

.i

Should be prepared by the user

.s

Figure 2.2 Processing Flow

## Input/Output File

The files input to and output from the ccimp are described here.

Any file name acceptable in the host environment is specifiable as a file name that the user specifies for the ccimp. However, "-" cannot be used as the initial character of the file name in the ccimp because "-" is assumed to be specification of an option.

### Input File

#### Types of input file

In the ccimp, the following files can be specified as input files.

The driver of the ccimp determines the file type from the extension and starts processing from the corresponding phase.

|  |  |
| --- | --- |
| .c | C source file  When a C source file is specified as the input file, it is compiled and then assembled in principle. |
| .s | Assembly source file  When an assembly source file is specified as the input file, it is assembled. |

\* Each file extension is not case-sensitive.

\* When a file with no extension or an extension not listed above is specified as the input file, it is regarded as a C source file. However, by -input=lang option, file type of the input file can also be specified.

#### C source file

The ccimp accepts C source files conforming to C99. Functions can be extended using #pragma directives.

The ASCII code, EUC, SJIS, and UTF-8 (with or without BOM) can be used in a C source file. However, the characters that can be handled by Japanese Windows in character codes of EUC (JP), SJIS and UTF-8 are as follows:

• JISX 0201 (JIS Roman characters, half-width kana)

• JIS X 0208: 1977 (JIS No. 1/2 level kanji)

Both "\n" and "\r\n" can be used as a newline character.

For details on language specifications of a C source file acceptable by the ccimp, refer to "Language Specifications of ccimp".

#### Assembly source file

The shader assembler accepts assembly source files which consist of shader instructions and pseudo instructions. For details on language specifications of assembly programs acceptable by the shader assembler, refer to "Specifications of Shader Assembler".

### Output File

#### Types of output file

The ccimp outputs the following files.

|  |  |
| --- | --- |
| .i | Source file after preprocessing |
| .s | Assembly source file |
| .o | Object file (relocatable file) |
| .out | Object file (executable file) |

The extension including the file name can be changed by using an option.

#### Object file

The format of an object file that is output by the ccimp is ELF. The debugging information conforms to the DWARF3 specifications.

#### Temporary file

The ccimp creates a work folder in the course of processing and generates temporary files for internal use. The location for creating a work folder is determined according to the priority order shown below.

1. Folder specified by the environment variable TEMP

2. Folder specified by the environment variable TMP

3. Current folder

### Command Line Format

ccimp[△option]...△file-name[△file-name or option]...

[ ] : Indicates that the item can be omitted.

... : Indicates that the item in the preceding [ ] can be specified one or more times.

△ : Indicates one or more spaces.

The ccimp returns 0 as the normal completion state (including cases where warnings were issued) or a non-zero value for an error.

When the -E, -S, or -c option is not specified, the default operation is compilation, assembly, and linkage when a C source file is input. The default operation is assembly and linkage when an assembly source file is input.

Multiple files can be specified as input files, and files of different types can be input together.

When multiple input files are specified, each of the source files is compiled and assembled in the order of specification in the command line, and a single executable object file is generated by linking the object file for each source file.

### Options

* Option characters are case-sensitive.
* For an option that requires a numerical value as a parameter, a decimal number or a hexadecimal number that starts with "0x" ("0X") can be specified.
* For an option that requires a path name as a parameter, an absolute path or a relative path can be specified.
* For an option that requires a file name as a parameter, a file name including its path (absolute path or relative path) can be specified. Any extension can be used. For a file name without a path or with a relative path, a path from the current folder is assumed.

#### List of Options

Legend:

[ ] : Indicates that the item can be omitted.

... : Indicates that the item in the preceding "[ ]" can be specified one or more times.

△ : Indicates one or more spaces.

{} : Indicates that a choice can be made from the items delimited by "|" in "{ }".

|  |  |
| --- | --- |
| -V | Print version and copyright. |
| -v | Displays the execution state. (Not open to users) |
| -help | Print help message. |
| -o [△] filename | Specify output file name. |
| -E | Execute preprocessing only. |
| -S | Output assembly files and quit. |
| -c | Output object files and quit. |
| -g | Generate debugging information. |
| -D [△] name [= value ] | Define preprocessor or assembler symbol 'name'. |
| -U [△] name | Invalidate definition of symbol 'name' previously defined with -D. |
| -I [△] directory | Add extra folder to include file search path. |
| -preincude = file [, file] ... | Specify pre-include file. |
| -cpu = {v3m2 | v3h | v3u} | Specify shader processor type. |
| -O [{ nothing | default | speed }] | Specify optimization level. |
| -Oinline [= num] | Specify performing of the in-line development. |
| -Onoinline | Specify not performing of the in-line development. |
| -Ounroll = num | Specify unrolling rate. |
| -Oschedule = {on | off} | Handle of instruction scheduling optimization. |
| -Ousecalla = {on | off} | Handle of CALLA instruction generation. |
| -Oifconversion = {on | off} | Handle of ifconversion. |
| -Ouse -anchor-symbol = {on | off} | Handle of optimization to reduce the number of symbolic address calculations. |
| -Ouselps = {on | off} | Handle of LPS/LPE instructions generation. |
| -num-uniform = num | Specify number of available uniform entries. |
| -num-implicit-uniform = <num> | Specify number of available implicit uniform entries. |
| -align-code[={function|innermostloop|loop|all}] | Specify alignment of branch destination address. |
| -stack-section = { lwm | impc } | Specify memory area of stack. |
| -round = { zero | nearest } | Specify round mode for floating-point constant. |
| -denormalize = {on | off } | Handles denormalized numbers as zeros or they are. |
| -no-warning = {num [, num] | num1-num2} | Disable warning with number 'num'. |
| -error = {num [, num] | num1-num2} | Appoint a warning message to consider to be an error. |
| -input- lang = { none | c | asm } | Specify language of following input files. |
| -input-charset = {none | sjis | euc-jp | utf8} | Specify encoding of multi-byte character sequence. |
| -init -regs-on-entry | Initialize registers R0, R1, R2, CR8 and CR9 to 0 in the entry function. |
| -asmopt = arg [, arg] ... | Pass option to assembler. |
| -lnkopt = arg | Pass option to linker. |
| -feopt = arg | Specifies an rcfrt option. (Not open to users) |
| -beopt = arg | Specifies an rcbackimp option. (Not open to users) |
| -drvopt = arg | Specifies a driver option. (Not open to users) |
| @ | Read command-file 'file'. |

#### Detailed Descriptions of Options

-V

* Function

Displays the version information.

* Format

-V

* Default behavior

The compiler compiles the input files without displaying the version information.

* Description

This option is used to output the version information to stderr without compiling files.

This option displays the versions of the compiler and assembler.

This option can be specified multiple times (even if specified multiple times, the operation is the same as when it is specified only once).

-v (Not open to users)

* Function

Displays the execution state.

* Format

-v

* Default behavior

The compiler does not display the execution state.

* Description

This option is used to display the state of compiler execution.

This option can be specified multiple times (even if specified multiple times, the operation is the same as when it is specified only once).

-help

* Function

Displays help information regarding options.

* Format

-help

* Default behavior

The compiler does not display help information regarding options.

* Description

This option is used to output help information regarding options to stderr without compiling files.

This option can be specified multiple times (even if specified multiple times, the operation is the same as when it is specified only once).

-o

* Function

Specifies the name of the output file.

* Format

-o[△]filename

* Default behavior
* When the -E option is specified

The output file name will be the input file name with its extension replaced with ".i".

* When the -S option is specified

The output file name will be the input file name with its extension replaced with ".s".

* When the -c option is specified

The output file name will be the input file name with its extension replaced with ".o".

* When the -E, -S, or -c option is not specified

The output file name will be the first-specified input file name with its extension replaced with ".out".

The output destination folder will be the current folder.

* Description

This option is used to specify filename as the output file name. If an existing file has the same name as filename, it will be overwritten.

* When this option is specified together with the -E option

The specified file name is assumed to be the name for the resultant file generated by preprocessing the input file.

An error will occur if the -o option is specified when multiple input files are specified, or if the -o option is specified multiple times.

* When this option is specified together with the -S option

The specified file name is assumed to be the name of the assembly source file.

An error will occur if the -o option is specified when multiple input files are specified, or if the -o option is specified multiple times.

* When this option is specified together with the -c option

The specified file name is assumed to be the name of the relocatable object file.

An error will occur if the -o option is specified when multiple input files are specified, or if the -o option is specified multiple times.

* When the -E, -S, or -c option is not specified

The specified file name is assumed to be the name of the object file.

An error will occur if the -o option is specified multiple times.

-E

* Function

Executes preprocessing only.

* Format

-E

* Default behavior

The compiler continues processing even after preprocessing. The preprocessed C source file is not output.

* Description

This option is used to execute only preprocessing for the input C source file and output the results to a file.

The output file name will be the input file name with its extension replaced with ".i", but the user can name the file by specifying this option and the -o option together.

This option can be specified multiple times (even if specified multiple times, the operation is the same as when it is specified only once).

-S

* Function

Ends processing before assembly.

* Format

-S

* Default behavior

The compiler continues processing.

* Description

The compiler ends processing after compilation is completed and does not proceed to execution of the assembler and subsequent modules. The assembly source file is output under the input file name with its extension replaced with ".s", but the user can name the file by specifying this option and the -o option together.

This option can be specified multiple times (even if specified multiple times, the operation is the same as when it is specified only once).

-c

* Function

Ends processing before linker.

* Format

-c

* Default behavior

The compiler executes processing of the linker.

* Description

The compiler ends processing after assembly is completed and does not proceed to execution of the linker. The relocatable object file is output under the input file name with its extension replaced with ".o", but the user can name the file by specifying this option and the -o option together.

This option can be specified multiple times (even if specified multiple times, the operation is the same as when it is specified only once).

-cpu

* Function

Generate code for the specified cpu.

* Format

-cpu={v3m2|v3h|v3u}

* Default behavior

An error will occur.

* Description

Generate code for the specified cpu.

* v3m2

Generate code for V3M2.

Define the macros \_\_IMP\_X5\_U\_\_ and \_\_RENESAS\_VERSION\_\_ = 0x03000000.

* v3h

Generate code for V3H.

Define the macros \_\_IMP\_X5\_U\_\_ and \_\_RENESAS\_VERSION\_\_ = 0x03000000.

* v3u

Generate code for V3U.

Define the macros \_\_IMP\_X6\_U\_\_ and \_\_RENESAS\_VERSION\_\_ = 0x03000000.

When multiple -cpu options are specified, when parameters are omitted, or when incorrect parameters are specified, an error occurs.

-g

* Function

Outputs debugging information.

* Format

-g

* Default behavior

The compiler does not output debugging information.

* Description

The compiler outputs information for debugging source code on e2 studio for ADAS.

This option can be specified multiple times (even if specified multiple times, the operation is the same as when it is specified only once).

-D

* Function

Defines a preprocessor macro.

* Format

-D[△]name[=value]

* Default behavior

None.

* Description

This option is used to define name as a preprocessor macro. This option has the same effect as writing "#define name value" at the beginning of the source file. When =value is omitted, value is assumed to be 1.

This option can be specified multiple times.

If both –D and -U are specified for a single preprocessor macro, the last option specified in the command line is valid.

-U

* Function

Invalidates a macro definition.

* Format

-U[△]name

* Default behavior

None.

* Description

This option is used to invalidate definition of the preprocessor macro name specified through the –D option or the predefined macro in the compiler. This option has the same effect as writing "#undef name" at the beginning of the source file; that is, -U cannot invalidate the definition of "#define name value" in the source file.

This option can be specified multiple times.

-I

* Function

Specifies a folder to search for header files (for the preprocessor).

* Format

-I[△]path

* Default behavior

The compiler searches in the following order.

1. Folder containing the source file

2. System header file folder (not open to users)

* Description

This option is used to specify a folder to search for the files to be included through the preprocessor directive #include.

Folders are searched in the following order.

1. Folder containing the source file (if the files to be included are enclosed in double quotation marks)

2. Folder specified through the –I option

3. System header file folder

A warning will be output if the folder specified with path is not found.

This option can be specified multiple times.

-preinclude

* Function

Specifies files to be included at the beginning of the current C source file.

* Format

-preinclude=file[,file]...

* Default behavior

The compiler includes no files at the beginning of the current C source file.

* Description

This option is used to specify C source files to be included at the beginning of the current C source file. When multiple files are specified, they are included in the order of specification in the command line.

An error will be output if the file specified with file is not found.

This option can be specified multiple times.

-O

* Function

Specifies the optimization level.

* Format

-O[{default|speed|nothing}]

* Default behavior

Same operation as when –Odefault is specified.

* Description

This option is used to specify the optimization level.

The following strings can be specified as a level. When the level specification is omitted (-O), -Ospeed is assumed.

\* The meaning of each level is the same as that in CC-RH. (Not open to users)

* default

Optimizes the code within the extent that will not greatly affect debugging.

* speed

Optimizes the code with precedence on execution speed.

* nothing

Stops all optimization processes.

If multiple optimization levels are specified, the last level specified in the command line is valid.

If the optimization level is specified for -Onothing or -Odefault, other optimization options cannot be specified.

-Oinline

* Function

Automatically executes inline expansion of functions.

* Format

-Oinline[=num]

* Default behavior

If -Ospeed is specified, num=100 is assumed.

* Description

This option is used to execute automatic inline expansion of functions. If -Onething or -Odefault is specified, it becomes invalid at the time of specification.

However, functions with #pragma noinline specified do not perform inline expansion.

\* The functions specified with #pragma inline are always expanded inline regardless of the –Oinline specification.

A value from 0 to 65535 can be specified for num.

Use num to specify the maximum increasing rate (%) of the function size up to which inline expansion is done.

When =num is omitted, num=100 is assumed.

When –Oinline is specified multiple times or both –Oinline and –Onoinline are specified together, the last –Oinline or –Onoinline specified in the command line is valid.

-Onoinline

* Function

Suppresses automatic inline expansion of functions.

* Format

-Onoinline

* Default behavior

The compiler does not suppress automatic inline expansion of functions.

* Description

This option is used to suppress automatic inline expansion of functions.

Note, however, that the ccimp attempts to expand functions for which #pragma inline is specified, regardless of whether the -Onoinline option is specified.

When both -Oinline and -Onoinline are specified together, the last -Oinline or -Onoinline specified in the command line is valid.

-Ounroll

* Function

Specifies loop expansion.

* Format

-Ounroll[=num]

* Default behavior

If -Ospeed is specified, num=4 is assumed.

* Description

This option is used to select loop expansion. If -Onething or -Odefault is specified, it becomes invalid at the time of specification.

A value from 0 to 32 can be specified for num, which is the maximum number of times to expand loops.

When 0 or 1 is specified, 1 loop expansion is performed.

An error will occur if a value other than 0 to 32 is specified.

This option can be specified multiple times, but only the last option specified in the command line is valid.

-Oschedule

* Function

Controls optimization with instruction reordering.

* Format

-Oschedule={on|off}

* Default behavior

If -Ospeed is specified, on is assumed. If -Onething or -Odefault is specified, it becomes invalid at the time of specification.

* Description

This option is used to select whether to reorder instructions for optimization.

* on

Reorders instructions for optimization.

* off

Does not reorder instructions for optimization.

This option can be specified multiple times, but only the last option specified in the command line is valid.

-Ousecalla

* Function

Controls optimization with the CALLA instruction.

* Format

-Ousecalla={on|off}

* Default behavior

If -Ospeed is specified, on is assumed.

* Description

The compiler controls whether to perform optimization that generates the CALLA instruction for a function call. If -Onething or -Odefault is specified, it becomes invalid at the time of specification.

* on

Performs optimization that generates the CALLA instruction for a function call.

* off

Does not generate the CALLA instruction.

This option can be specified multiple times, but only the last option specified in the command line is valid.

-Oifconversion

* Function

Controls optimization with conditional instructions.

* Format

-Oifconversion={on|off}

* Default behavior

If -Ospeed is specified, on is assumed.

* Description

This option is used to select whether to generate conditional instructions for optimization. If -Onething or -Odefault is specified, it becomes invalid at the time of specification.

* on

Generates conditional instructions for optimization.

* off

Does not generate conditional instructions.

This option can be specified multiple times, but only the last option specified in the command line is valid.

-Ouse-anchor-symbol

* Function

Controls optimization of static variable access.

* Format

-Ouse-anchor-symbol={on|off}

* Default behavior

If -Ospeed is specified, on is assumed.

* Description

The compiler controls whether to perform optimization of static variable access. If -Onething or -Odefault is specified, it becomes invalid at the time of specification.

* on

Performs optimization of static variable access.

* off

Does not perform optimization of static variable access.

This option can be specified multiple times, but only the last option specified in the command line is valid.

-Ouselps

* Function

Handles the LPS/LPE instructions generation.

* Format

-Ouselps ={on|off}

* Default behavior

If -Ospeed is specified, on is assumed.

* Description

The compiler controls whether to enable the generation of LPS/LPE instruction. If -Onething or -Odefault is specified, it becomes invalid at the time of specification.

* on

Enable the generation of LPS/LPE instruction.

* off

Disable the generation of LPS/LPE instruction.

A list of specified whether or not in combination with c pu options in Table 2.4.2.1.

Table 2.4.2.1. List of combinations with cpu option

|  |  |  |
| --- | --- | --- |
| cpu option | Ouselps option | Specification possibility |
| - cpu = v3m2 | - Ouselps = off | OK |
| - cpu = v3m2 | - Ouselps = on | Error \* |
| - cpu = v3h | - Ouselps = off | OK |
| - cpu = v3h | - Ouselps = on | Error \* |
| - cpu = v3u | - Ouselps = off | OK |
| - cpu = v3u | - Ouselps = on | OK |

It becomes effective when -Ospeed is specified before -Ouselps. If -Ospeed is specified after -Ouselps, or -Ospeed is not specified, the -Ouselps specification is ignored and no error occurs.

-num-uniform

* Function

Specifies the number of entries in the uniform area.

* Format

-num-uniform=num

* Default behavior

The compiler recognizes that 64 has been specified for num.

* Description

This option is used to specify the number of entries in the uniform area which is available in each compiling unit.

In some cases, the compiler will create the constant data in the uniform area which is used for programs within the range of the number of specified entries. However, since data of const-qualified global variables must be allocated in the uniform area, an error will occur if the number of const-qualified global variables exceeds the number of specified entries.

A value from 0 to 64 can be specified for num.

This option can be specified multiple times, but only the last option specified in the command line is valid.

-num-implicit-uniform

* Function

Specifies the number of entries in the uniform area to be used for implicit.

* Format

-num-implicit-uniform=<num>

* Default behavior

The number of compilers that use the Uniform area implicitly is the number obtained by subtracting the number of const global variables from the specified value of -num\_uniform

* Description

Specify the number of uniform regions to be used implicitly by the compiler.  
If this option is specified more than once, the last <num> specified will be valid.

* Num

An integer in the range 0 to -num-uniform.

When 0 is specified, constant values other than const global variables are not allocated to the Uniform area.

When the parameter is omitted, or an out-of-range parameter is specified, an error occurs.

-stack-section

* Function

Specifies the stack area section.

* Format

-stack-section={lwm|impc}

* Default behavior

The compiler assumes that lwm is specified.

* Description

This option specifies the section for which the stack area is to be allocated.

This option can be specified multiple times, but only the last option specified in the command line is valid.

-align-code

* Function

Specifies alignment of branch destination addresses.

* Format

-align-code[=mode]

* Default behavior

Alignment of branch destination addresses is not specified.

* Description

32-byte alignment is applied to branch addresses as specified by mode.

For mode, one of the following values can be specified. If a value other than the following ones is specified, an error results.

* function

The beginning address of the function is aligned by 32 bytes.

* loop

The beginning addresses of the function and all loops are aligned by 32 bytes.

* innermostloop

The beginning addresses of the function and innermost loop are aligned by 32 bytes.

* all

The beginning address of the function and all branch destination addresses are aligned by 32 bytes.

* If =mode is omitted

The beginning address of the function is aligned by 32 bytes. (This is the same behavior as when function is specified.)

-round (Not open to users)

* Function

Specifies the rounding mode for floating-point operation.

* Format

-round={zero|nearest}

* Default behavior

The compiler rounds in the round-to-nearest mode (-round=nearest).

* Description

This option is used to select the rounding mode for floating-point operation.

* zero

Rounds in the round-to-zero mode.

* nearest

Rounds in the round-to-nearest mode.

This option can be specified multiple times, but only the last option specified in the command line is valid.

Note that this option does not change the rounding mode for floating-point operation at run time.

-denormalize (Not open to users)

* Function

Specifies handling of denormalized floating-point numbers.

* Format

-denormalize={on|off}

* Default behavior

The compiler does not handle denormalized numbers as 0 (-denormalize=on).

* Description

This option is used to determine how to handle denormalized floating-point numbers written in the code.

* on

Handles denormalized numbers as they are (as denormalized numbers).

* off

Handles denormalized numbers as 0.

This option can be specified multiple times, but only the last option specified in the command line is valid.

-no-warning

* Function

Suppresses output of specific warning messages.

* Format

-no-warning={num[,num]...|num1-num2}

* Default behavior

The compiler outputs all warning messages.

* Description

This option is used to suppress output of specific warning messages.

Use num to specify an error number. A sequence of numbers can be specified by using a hyphen between the lower limit (num1) and upper limit (num2); this assumes that all error numbers in the range are specified. For this option, specify the lower five digits in the seven-digit number that follows "W" in the error number.

This option can only control the warning messages output by the compiler.

This option can be specified multiple times.

-error

* Function

Outputs specific warning messages as errors.

* Format

-error={num[,num]...|num1-num2}

* Default behavior

The compiler outputs all warning messages as they are (as warning messages).

* Description

This option is used to select specific warning messages to be output as errors.

Use num to specify an error number. A sequence of numbers can be specified by using a hyphen between the lower limit (num1) and upper limit (num2); this assumes that all error numbers in the range are specified. For this option, specify the lower five digits in the seven-digit number that follows "W" in the error number.

This option can only control the warning messages output by the compiler.

This option can be specified multiple times.

-asmopt

* Function

Specifies an assembler option.

* Format

-asmopt=arg[,arg]

* Default behavior

The compiler assumes that no options in the command line should be passed to the assembler.

* Description

This option is used to pass arg to the assembler as an assembler option. To specify an assembler option that requires a parameter, specify the parameter following the option with delimiting them with a comma (,).

The assembler will output an error if the specified assembler option does not exist.

This option can be specified multiple times.

-lnkopt

* Function

Specifies a linker option.

* Format

-lnkopt=arg

* Default behavior

The compiler assumes that no options in the command line should be passed to the linker.

* Description

This option is used to pass arg to the linker as a linker option.

The linker will output an error if the specified linker option does not exist.

This option can be specified multiple times.

-input-lang

* Function

Recognize input source type (C / Assembly ) after this option specification with type according to option specification instead of extension

* Format

-input-lang={none|c|asm}

* Default behavior

Determine the input source type by extension.

* Description

Recognize input source type (C / Assembly) after specification of this option not by extension but by type according to option specification. This option can be specified for each input source. If you specify multiple for one input source, the option specified later becomes effective.

* none

Specify language of input source file based on the extension.

* c

Specify language of input source file as a C source file.

* asm

Specify language of input source file as an assembly source file.

If parameter is omitted or incorrect parameter is specified, an error occurs.

-input-charset

* Function

Specifies the Japanese character coding.

* Format

-input-charset={none|sjis|euc-jp|utf8}

* Default behavior

Assumes sjis when a Japanese OS is used, and assumes none in other-language OS's.

* Description

This option is used to select character coding to be used for Japanese comments in the input file. Correct operation is not guaranteed if the specified character coding differs from the actual character coding used in the input file.

An error will occur if any character coding other than those listed below is specified.

* none

Does not process the Japanese character coding.

* euc-jp

EUC (Japanese)

* sjis

Shift-JIS

* utf8

UTF-8

When this option is specified multiple times, the last option specified in the command line is valid.

-init-regs-on-entry

* Function

Insert an instruction sequence that initializes the contents of registers R0 / R1 / R2 / CR8 / CR9 to 0 at the beginning of the entry function

* Format

-init-regs-on-entry

* Default behavior

The contents of registers R0 / R1 / R2 / CR8 / CR9 become uninitialized.

* Description

At the beginning of the entry function, an instruction sequence for initializing the contents of the registers R0 / R1 / R2 / CR8 / CR9 to 0 is inserted.

-feopt (Not open to users)

* Function

Specifies an rcfrt option.

* Format

-feopt=arg

* Default behavior

The compiler assumes that no options in the command line should be passed to rcfrt.

* Description

This option is used to pass arg to rcfrt as an option.

This option can be specified multiple times.

-beopt (Not open to users)

* Function

Specifies an rcbackimp option.

* Format

-beopt=arg

* Default behavior

The compiler assumes that no options in the command line should be passed to rcbackimp.

* Description

This option is used to pass arg to rcbackimp as an option.

This option can be specified multiple times.

@

* Function

Specifies a subcommand file.

* Format

@file

* Default behavior

The compiler recognizes only the options and files specified in the command line.

* Description

This option is used to handle file as a subcommand file.

A subcommand file contains the options and file names to be specified for the ccimp command. Specifying this file has the same effect as when these options and files are specified in the command line.

This option can be specified multiple times.

# C99 Library

## Standard C99 Library

### Overview of C99 Library

This section describes the specifications of the C99 library functions, which can be used generally in C/C++ programs. This section gives an overview of the library configuration, and describes the layout and the terms used in this library function description.

##### Library Types

A library implements standard processing such as input/output and string handling in the form of C/C++ language functions. Libraries can be used by including standard include files for each unit of processing.

Standard include files contain declarations for the corresponding libraries and definitions of the macro names necessary to use them.

Table 3.1 shows the various library types and the corresponding standard include files.

Table 3.1 Library Types and Corresponding Standard Include Files

|  |  |  |
| --- | --- | --- |
| Library Type | Description | Standard Include File |
| Program diagnostics | Outputs program diagnostic information. | <assert.h> |
| Errors | Support reporting of error conditions | <errno.h> |
| Floating-point environment | Supports access to floating-point environment. | <fenv.h> |
| Characteristics of floating types | Defines various limit values relating to the limits of floating-point numbers. | <float.h> |
| Mathematics | Performs numerical calculations such as trigonometric functions. | <math.h>  <xtgmath.h>  <xmath.h>  <ymath.h> |
| Common definitions | Defines macro names used in common by the standard include files. | <stddef.h> |
| General utilities | Performs C program standard processing such as storage area management. | <stdlib.h> |
| String handling | Performs string comparison, copying, etc. | <string.h> |
| Dinkum | Defines Dinkum C99 library | <yvals.h>  <xkeycheck.h> |
| Renesas library | Renesas macro define include file | <macro.h> |

In addition to the above standard include files, standard include files consisting solely of macro name definitions, shown in Table 3.2, are provided to improve programming efficiency.

Table 3.2 Standard Include Files Comprising Macro Name Definitions

|  |  |
| --- | --- |
| Standard Include File | Description |
| <limits.h> | Defines various limit values relating to compiler internal processing. |
| <stdbool.h> | Defines macros relating to logical types and values. |
| <iso646.h> | Declares macros conforming with ISO 646 standard |

##### Organization of Library Part

The organization of the library part of this manual is described below.

Library functions are categorized according to the corresponding standard include file, and descriptions are given for each standard include file. For each category, there is first a description relating to the macro names and function declarations defined in the standard include file (Figure 3.1), followed by a description of each function (Figure 3.2).

Figure 3.1 shows the standard include file description layout, and Figure 3.2, the function description layout.



Figure 3.1 Layout of Standard Include File Description

Function type and name (return value and parameters) Functional overview

Description: Describes the library function.

Header file: Shows the name of standard include file that contains this function

declaration.

Return value: Normal: Shows the return value when the library function ends normally.

Abnormal: Shows the return value when the library function ends

abnormally.

Parameters: Indicates the meanings of the parameters.

Example: Describes the calling procedure.

Error conditions: Conditions for the occurrence of errors that cannot be determined from the

return value in library function processing.

If such an error occurs, the value defined in each compiler for the error

type is set in **errno**\*.

Remarks: Provides summplementary information or notes on usage.

Implementation define: Describes the processing method in this compiler

Figure 3.2 Layout of Function Description

Note: **errno** is a variable that stores the error type if an error occurs during execution of a library function. See section 3.1.1.3, <errno.h>, for details.

##### Terms Used in Library Function Descriptions

###### Stream input/output

In data input/output, it would lead to poor efficiency if each call of an input/output function, which handles a single character, drove the input/output device and the OS functions. To solve this problem, a storage area called a buffer is normally provided, and the data in the buffer is input or output at one time.

From the viewpoint of the program, on the other hand, it is more convenient to call input/output functions for each character.

Using the library functions, character-by-character input/output can be performed efficiently without awareness of the buffer status within the program by automatically performing buffer management.

Those library functions enable a programmer to write a program considering the input/output as a single data stream, making the programmer be able to implement data input/output efficiently without being aware of the detailed procedure. Such capability is called stream input/output.

###### Functions and macros

There are two library function implementation methods: functions and macros.

A function has the same interface as an ordinary user-written function, and is incorporated during linkage. A macro is defined using a **#define** statement in the standard include file relating to the function.

The following points must be noted concerning macros:

* Macros are expanded automatically by the preprocessor, and therefore a macro expansion cannot be invalidated even if the user declares a function with the same name.
* If an expression with a side effect (assignment expression, increment and decrement) is specified as a macro parameter, its result is not guaranteed.

Example: Macro definition of **MACRO** that calculates the absolute value of a parameter is as follows:

If the following definition is made:

#define MACRO(a) ((a) >= 0 ? (a) : -(a))

and if

X=MACRO(a++)

is in the program, the macro will be expanded as follows:

X = ((a++) >= 0 ? (a++) : -(a++))

a will be incremented twice, and the resultant value will be different from the absolute value of the initial value of a.

###### NULL

This is the value indicating that a pointer is not pointing at anything. The name **NULL** is defined in the **<stddef.h>**, **<stdlib.h>** and **<string.h>** standard include file.

###### Return code

With some library functions, a return value is used to determine the result (such as whether the specified processing succeeded or failed). In this case, the return value is called the return code.

###### Text files and binary files

Many systems have special file formats to store data. To support this facility, library functions have two file formats: text files and binary files.

* Text files

A text file is used to store ordinary text, and consists of a collection of lines. In text file input, the new-line character (\n) is input as a line separator. In output, output of the current line is terminated by outputting the new-line character (\n). Text files are used to input/output files that store standard text for each system. With text files, characters input or output by a library function do not necessarily correspond to a physical stream of data in the file.

* Binary files

A binary file is configured as a row of byte data. Data input or output by a library function corresponds to a physical list of data in the file.

###### Standard input/output files

Files that can be used as standard by input/output library functions by default without preparations such as opening file are called standard input/output files. Standard input/output files comprise the standard input file (**stdin**), standard output file (**stdout**), and standard error output file (**stderr**).

* Standard input file (**stdin**)

Standard file to be input to a program.

* Standard output file (**stdout**)

Standard file to be output from a program.

* Standard error output file (**stderr**)

Standard file for storing output of error messages, etc., from a program.

##### Notes on Use of Libraries

The contents of macros defined in a library differ for each compiler.

When a library is used, the behavior is not guaranteed if the contents of these macros are redefined.

With libraries, errors are not detected in all cases. The behavior is not guaranteed if library functions are called in a form other than those shown in the descriptions in the following sections.

#### <assert.h>

Adds diagnostics into programs.

|  |  |  |
| --- | --- | --- |
| Type | Definition Name | Description |
| Function (macro) | assert | Adds diagnostics into programs. |

To invalidate the diagnostics defined by **<assert.h>**, define macro name **NDEBUG** with a **#define** statement (**#define** **NDEBUG**) before including **<assert.h>**.

Note: If a **#undef** statement is used for macro name **assert**, the result of subsequent **assert** calls is not guaranteed.

**void assert (*scalar* expression) Diagnostics**

Description: Adds diagnostics into programs.

Header file: <assert.h>

Parameters: expression Expression to be evaluated.

Example: #include <assert.h>

int expression;

assert (expression);

Remarks: When **expression** is true, the **assert** macro terminates processing without returning a value. If **expression** is false, it outputs diagnostic information to the standard error file in the form defined by the compiler, and then calls the **abort** function.

The diagnostic information includes the parameter's program text, source file name, and source line numbers.

#### <errno.h>

Defines the value to be set in **errno** when an error is generated in a library function.

The following macro names are all implementation-defined.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Variable  (macro) | errno |  | **int** type variable. An error number is set when an error is generated in a library function. |
| Constant  (macro) | ERANGE | 34 | Indicates the value stored in errno on a range error. |
| EDOM | 33 | Indicates the value stored in errno on a domain error |

#### <fenv.h>

Provides access to the floating-point environment.

The following macros and functions are all implementation-defined.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
|  | FE\_DOWNWARD  FE\_TONEAREST  FE\_TOWARDZERO  FE\_UPWARD | 3  0  1  2 | Indicates the values (macros) of the floating-point rounding direction. |
| Function | fegetround |  | Gets the rounding direction. |

int fegetround(void) Getting Rounding Direction

Description: Gets the current rounding direction.

Header file: <fenv.h>

Return values: Normal: 0

Abnormal: Negative value when there is no rounding direction macro value or the rounding

direction cannot be determined

Example: #include <fenv.h>

#pragma STDC FENV\_ACCESS ON

int ret = fegetround();

Remarks: None

#### <float.h>

Defines various limits relating to the internal representation of floating-point numbers.

The following macro names are all implementation-defined.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Constant  (macro) | FLT\_RADIX | 2 | Indicates the radix in exponent representation. |
| FLT\_ROUNDS | 1 | Indicates whether or not the result of an add operation is rounded off.  The meaning of this macro definition is as follows:   * When result of add operation is rounded off: Positive value * When result of add operation is rounded down: 0 * When nothing is specified: -1   The rounding-off and rounding-down methods are implementation-defined. |
| FLT\_MAX | 3.4028234663852886e+38F | Indicates the maximum value that can be represented as a **float** type floating-point value. |
| DBL\_MAX | 3.4028234663852886e+38F | Indicates the maximum value that can be represented as a **double** type floating-point value. |
| LDBL\_MAX | 1.7976931348623158e+308 | Indicates the maximum value that can be represented as a **long double** type floating-point value. |
| FLT\_MAX\_EXP | 128 | Indicates the power-of-radix maximum value that can be represented as a **float** type floating-point value. |
| DBL\_MAX\_EXP | 128 | Indicates the power-of-radix maximum value that can be represented as a **double** type floating-point value. |
| LDBL\_MAX\_EXP | 16384 | Indicates the power-of-radix maximum value that can be represented as a **long double** type floating-point value. |
| FLT\_MAX\_10\_EXP | 38 | Indicates the power-of-10 maximum value that can be represented as a **float** type floating-point value. |
| DBL\_MAX\_10\_EXP | 38 | Indicates the power-of-10 maximum value that can be represented as a **double** type floating-point value. |
| LDBL\_MAX\_10\_EXP | 308 | Indicates the power-of-10 maximum value that can be represented as a **long double** type floating-point value. |
| FLT\_MIN | 1.175494351e38F | Indicates the minimum positive value that can be represented as a **float** type floating-point value. |
| DBL\_MIN | 1.175494351e-38F | Indicates the minimum positive value that can be represented as a **double** type floating-point value. |
| LDBL\_MIN | 2.2250738585072014e-308 | Indicates the minimum positive value that can be represented as a **long double** type floating-point value. |

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Constant  (macro) | FLT\_MIN\_EXP | -125 | Indicates the power-of-radix minimum value of a floating-point value that can be represented as a **float** type positive value. |
| DBL\_MIN\_EXP | -125 | Indicates the power-of-radix minimum value of a floating-point value that can be represented as a **double** type positive value. |
| LDBL\_MIN\_EXP | -1074 | Indicates the power-of-radix minimum value of a floating-point value that can be represented as a **long double** type positive value. |
| FLT\_MIN\_10\_EXP | -37 | Indicates the power-of-10 minimum value of a floating-point value that can be represented as a **float** type positive value. |
| DBL\_MIN\_10\_EXP | -37 | Indicates the power-of-10 minimum value of a floating-point value that can be represented as a **double** type positive value. |
| LDBL\_MIN\_10\_EXP | -323 | Indicates the power-of-10 minimum value of a floating-point value that can be represented as a **long double** type positive value. |
| FLT\_DIG | 6 | Indicates the maximum number of digits in **float** type floating-point value decimal-precision. |
| DBL\_DIG | 6 | Indicates the maximum number of digits in **double** type floating-point value decimal-precision. |
| LDBL\_DIG | 15 | Indicates the maximum number of digits in **long double** type floating-point value decimal-precision. |
| FLT\_MANT\_DIG | 24 | Indicates the maximum number of mantissa digits when a **float** type floating-point value is represented in the radix. |
| DBL\_MANT\_DIG | 24 | Indicates the maximum number of mantissa digits when a **double** type floating-point value is represented in the radix. |
| LDBL\_MANT\_DIG | 53 | Indicates the maximum number of mantissa digits when a **long double** type floating-point value is represented in the radix. |
| DECIMAL\_DIG | 8 | Indicates the maximum number of digits of a floating-point value represented in decimal precision. |
| FLT\_EPSILON | 1.19209e-07 | Indicates the difference between 1 and the minimum value greater than 1 that can be represented in **float** type. |
| DBL\_EPSILON | 1.19209e-07 | Indicates the difference between 1 and the minimum value greater than 1 that can be represented in **double** type. |
| LDBL\_EPSILON | 1E-9 | Indicates the difference between 1 and the minimum value greater than 1 that can be represented in **long double** type. |

#### <limits.h>

Defines various limits relating to the internal representation of integer type data.

The following macro names are all implementation-defined.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Constant  (macro) | CHAR\_BIT | 8 | Indicates the number of bits in a **char** type value. |
| CHAR\_MAX | 127 | Indicates the maximum value that can be represented by a **char** type variable. |
| CHAR\_MIN | -128 | Indicates the minimum value that can be represented by a **char** type variable. |
| SCHAR\_MAX | 127 | Indicates the maximum value that can be represented by a **signed char** type variable. |
| SCHAR\_MIN | -128 | Indicates the minimum value that can be represented by a **signed char** type variable. |
| UCHAR\_MAX | 255U | Indicates the maximum value that can be represented by an **unsigned char** type variable. |
| SHRT\_MAX | 32767 | Indicates the maximum value that can be represented by a **short** type variable. |
| SHRT\_MIN | -32768 | Indicates the minimum value that can be represented by a **short** type variable. |
| USHRT\_MAX | 65535U | Indicates the maximum value that can be represented by an **unsigned short** type variable. |
| INT\_MAX | 2147483647 | Indicates the maximum value that can be represented by an **int** type variable. |
| INT\_MIN | -2147483648 | Indicates the minimum value that can be represented by an **int** type variable. |
| UINT\_MAX | 4294967295U | Indicates the maximum value that can be represented by an **unsigned int** type variable. |
| LONG\_MAX | 2147483647L | Indicates the maximum value that can be represented by a **long** type variable. |
| LONG\_MIN | -2147483648L | Indicates the minimum value that can be represented by a **long** type variable. |
| ULONG\_MAX | 4294967295U | Indicates the maximum value that can be represented by an **unsigned long** type variable. |

#### <math.h>

Performs various mathematical operations.

The following constants (macros) are all implementation-defined.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Constant  (macro) | HUGE\_VAL  HUGE\_VALF  HUGE\_VALL | INFINITY | Indicates the value for the function return value if the result of a function overflows. |
| INFINITY | INFINITY | Expanded to a **float**-type constant expression that represents positive or unsigned infinity. |
| NAN |  | Defined when **float**-type **qNaN** is supported. |
| FP\_INFINITE  FP\_NAN  FP\_NORMAL  FP\_SUBNORMAL  FP\_ZERO | 1  2  -1  -2  0 | These indicate exclusive types of floating-point values. |
| FP\_FAST\_FMA  FP\_FAST\_FMAF  FP\_FAST\_FMAL | 1  1  1 | Defined when the **Fma** function is executed at the same or higher speed than a multiplication and an addition with **double**-type operands. |
| FP\_ILOGB0  FP\_ILOGBNAN | -2147483648  2147483647 | These are expanded to an integer constant expression of the value returned by **ilogb** when they are 0 or not-a-number, respectively. |
| MATH\_ERRNO  MATH\_ERREXCEPT | 1  2 | These are expanded to integer constants 1 and 2, respectively. |
| math\_errhandling | 3 | Expanded to an int-type expression whose value is a bitwise logical OR of **MATH\_ERRNO** and **MATH\_ERREXCEPT**. |
| Type | float\_t |  | These are floating-point types having the same width as **float**, respectively. |

|  |  |  |
| --- | --- | --- |
| Type | Definition Name | Description |
| Constant  (macro) | fpclassify | Classifies argument values into not-a-number, infinity, normalized number, denormalized number, and 0. |
| isfinite | Determines whether the argument is a finite value. |
| isinf | Determines whether the argument is infinity. |
| isnan | Determines whether the argument is a not-a-number. |
| isnormal | Determines whether the argument is a normalized number. |
| signbit | Determines whether the sign of the argument is negative. |
| isgreater | Determines whether the first argument is greater than the second argument. |
| isgreaterequal | Determines whether the first argument is equal to or greater than the second argument. |
| isless | Determines whether the first argument is smaller than the second argument. |
| Islessequal | Determines whether the first argument is equal to or smaller than the second argument. |
| Islessgreater | Determines whether the first argument is smaller or greater than the second argument. |
| Function | acosf | Calculates the arc cosine of a floating-point number. |
| asinf | Calculates the arc sine of a floating-point number. |
| atanf | Calculates the arc tangent of a floating-point number. |
| atan2f | Calculates the arc tangent of the result of a division of two floating-point numbers. |
| cosf | Calculates the cosine of a floating-point radian value. |

|  |  |  |
| --- | --- | --- |
| Type | Definition Name | Description |
| Function | sinf | Calculates the sine of a floating-point radian value. |
| tanf | Calculates the tangent of a floating-point radian value. |
| coshf | Calculates the hyperbolic cosine of a floating-point number. |
| sinhf | Calculates the hyperbolic sine of a floating-point number. |
| tanhf | Calculates the hyperbolic tangent of a floating-point number. |
| expf | Calculates the exponential function of a floating-point number. |
| frexpf | Breaks a floating-point number into a [0.5, 1.0) value and a power of 2. |
| ldexpf | Multiplies a floating-point number by a power of 2. |
| logf | Calculates the natural logarithm of a floating-point number. |
| log10f | Calculates the base-ten logarithm of a floating-point number. |
| modff | Breaks a floating-point number into integral and fractional parts. |
| powf | Calculates a power of a floating-point number. |
| sqrtf | Calculates the positive square root of a floating-point number. |
| ceilf | Calculates the smallest integral value not less than or equal to the given floating-point number. |
| fabsf | Calculates the absolute value of a floating-point number. |
| floorf | Calculates the largest integral value not greater than or equal to the given floating-point number. |
| fmodf | Calculates the remainder of a division of two floating-point numbers. |
| acoshf | Calculates the hyperbolic arc cosine of a floating-point number. |
| asinhf | Calculates the hyperbolic arc sine of a floating-point number. |
| atanhf | Calculates the hyperbolic arc tangent of a floating-point number. |
| exp2f | Calculates the value of 2 raised to the power **x**. |
| expm1f | Calculates the natural logarithm raised to the power **x** and subtracts 1 from the result. |
| ilogbf | Extracts the exponent of **x** as a signed **int** value. |
| log1pf | Calculates the natural logarithm of the argument + 1. |
| log2f | Calculates the base-2 logarithm. |
| logbf | Extracts the exponent of **x** as a signed integer. |
| scalbnf  scalblnf | Calculates **x** × **FLT\_RADIXn**. |
| cbrtf | Calculates the cube root of a floating-point number. |
| hypotf | Raises each floating-point number to the power 2 and calculates the sum of the resultant values. |
| erff | Calculates the error function. |
| erfcf | Calculates the complementary error function. |
| lgammaf | Calculates the natural logarithm of the absolute value of the gamma function. |
| tgammaf | Calculates the gamma function. |

|  |  |  |
| --- | --- | --- |
| Type | Definition Name | Description |
| Function | nearbyintf | Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction. |
| rintf | Equivalent to **nearbyint** except that this function group may generate floating-point exception. |
| lrintf | Rounds a floating-point number to the nearest integer according to the rounding direction. |
| roundf | Rounds a floating-point number to the nearest integer in the floating-point representation. |
| lroundf | Rounds a floating-point number to the nearest integer. |
| truncf | Rounds a floating-point number to the nearest integer. |
| remainderf | Calculates remainder **x** **REM** **y** specified in the IEEE60559 standard. |
| remquof | Calculates the value having the same sign as **x/y** and the absolute value congruent modulo-2n to the absolute value of the quotient. |
| copysignf | Generates a value consisting of the given absolute value and sign. |
| nanf | **nan("n string")** is equivalent to **("NAN(n string)", (char\*\*) NULL)**. |
| nextafterf | Converts a floating-point number to the type of the function and calculates the representable value following the converted number on the real axis. |
| fdimf | Calculates the positive difference. |
| fmaxf | Obtains the greater of two values. |
| fminf | Obtains the smaller of two values. |
| fmaf | Calculates **(d1 \* d2) + d3** as a single ternary operation. |

Operation in the event of an error is described below.

(1) Domain error

A domain error occurs if the value of a parameter input to a function is outside the domain over which the mathematical function is defined. In this case, the value of **EDOM** is set in **errno**. The function return value in implementation-defined.

(2) Range error

A range error occurs if the result of a function cannot be represented as a value of the double type. In this case, the value of **ERANGE** is set in **errno**. If the result overflows, the function returns the value of **HUGE\_VAL**, **HUGE\_VALF**, or **HUGE\_VALL** with the same sign as the correct value of the function. If the result underflows, 0 is returned as the return value.

Notes: 1.

If there is a possibility of a domain error resulting from a **<math.h>** function call, it is dangerous to use the resultant value directly. The value of **errno** should always be checked before using the result in such cases.

Example:

.

.

.

1 x=asin(a);

2 if (errno==EDOM)

3 printf ("error\n");

4 else

5 printf ("result is : %lf\n",x);

.

.

.

In line 1, the arc sine value is computed using the **asin** function. If the value of argument a is outside the **asin** function domain [-1.0, 1.0], the **EDOM** value is set in **errno**. Line 2 determines whether a domain error has occurred. If a domain error has occurred, **error** is output in line 3. If there is no domain error, the arc sine value is output in line 5.

Notes: 2.

Whether or not a range error occurs depends on the internal representation format of floating-point types determined by the compiler. For example, if an internal representation format that allows an infinity to be represented as a value is used, **<math.h>** library functions can be implemented without causing range errors.

float acosf (float d) 　 Arc Cosine

Description: Calculates the arc cosine of a floating-point number.

Header file: <math.h>

Return values: Normal: Arc cosine of **d**

Abnormal: Domain error: Returns not-a-number.

Parameters: d Floating-point number for which arc cosine is to be computed

Example: #include <math.h>

float d, ret;

ret=acosf(d);

Error conditions: A domain error occurs for a value of **d** not in the range [-1.0, +1.0].

Remarks: The **acos** function returns the arc cosine in the range [0, π] by the radian.

float asinf (float d) 　 Arc Sine

Description: Calculates the arc sine of a floating-point number.

Header file: <math.h>

Return values: Normal: Arc sine of **d**

Abnormal: Domain error: Returns not-a-number.

Parameters: d Floating-point number for which arc sine is to be computed

Example: #include <math.h>

float d, ret;

ret=asinf(d);

Error conditions: A domain error occurs for a value of **d** not in the range [-1.0, +1.0].

Remarks: The **asin** function returns the arc sine in the range [-π/2, +π/2] by the radian.

float atanf (float d) 　 Arc Tangent

Description: Calculates the arc tangent of a floating-point number.

Header file: <math.h>

Return values: Arc tangent of **d**

Parameters: d Floating-point number for which arc tangent is to be computed

Example: #include <math.h>

float d, ret;

ret=atanf(d);

Remarks: The **atan** function returns the arc tangent in the range (-π/2, +π/2) by the radian.

float atan2f (float y, float x) 　 Arc Tangent after Division

Description: Calculates the arc tangent of the division of two floating-point numbers.

Header file: <math.h>

Return values: Normal: Arc tangent value when y is divided by **x**

Abnormal: Domain error: Returns not-a-number.

Parameters: x Divisor

y Dividend

Example: #include <math.h>

float x, y, ret;

ret=atan2f(y, x);

Error conditions: A domain error occurs if the values of both **x** and **y** are 0.0.

Remarks: The **atan2** function returns the arc tangent in the range (-π, +π) by the radian. The meaning of the **atan2** function is illustrated in Figure 3.3. As shown in the figure, the result of the **atan2** function is the angle between the X-axis and a straight line passing through the origin and point (**x**, **y**).

If **y** = 0.0 and **x** is negative, the result is π. If **x** = 0.0, the result is ±π/2, depending on whether **y** is positive or negative.



Figure 3.3 Meaning of atan2 Function

float cosf (float d) 　 Cosine

Description: Calculates the cosine of a floating-point radian value.

Header file: <math.h>

Return values: Cosine of **d**

Parameters: d Radian value for which cosine is to be computed

Example: #include <math.h>

float d, ret;

ret=cosf(d);

float sinf (float d) 　 Sine

Description: Calculates the sine of a floating-point radian value.

Header file: <math.h>

Return values: Sine of **d**

Parameters: d Radian value for which sine is to be computed

Example: #include <math.h>

float d, ret;

ret=sinf(d);

float tanf (float d) 　 Tangent

Description: Calculates the tangent of a floating-point radian value.

Header file: <math.h>

Return values: Tangent of **d**

Parameters: d Radian value for which tangent is to be computed

Example: #include <math.h>

float d, ret;

ret=tanf(d);

float coshf (float d) 　 Hyperbolic Cosine

Description: Calculates the hyperbolic cosine of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic cosine of **d**

Parameters: d Floating-point number for which hyperbolic cosine is to be computed

Example: #include <math.h>

float d, ret;

ret=coshf(d);

float sinhf (float d) 　 Hyperbolic Sine

Description: Calculates the hyperbolic sine of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic sine of **d**

Parameters: d Floating-point number for which hyperbolic sine is to be computed

Example: #include <math.h>

float d, ret;

ret=sinhf(d);

float tanhf (float d) 　 Hyperbolic Tangent

Description: Calculates the hyperbolic tangent of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic tangent of **d**

Parameters: d Floating-point number for which hyperbolic tangent is to be computed

Example: #include <math.h>

float d, ret;

ret=tanhf(d);

float expf (float d) Exponential Function

Description: Calculates the exponential function of a floating-point number.

Header file: <math.h>

Return values: Exponential function value of **d**

Parameters: d Floating-point number for which exponential function is to be computed

Example: #include <math.h>

float d, ret;

ret=expf(d);

float frexpf (float value, int \* exp) Breaking Floating-Point Number into Mantissa and Exponent

Description: Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.

Header file: <math.h>

Return values: If **value** is 0.0: 0.0

If **value** is not 0.0: Value of **ret** defined by ret \* 2value pointed to by exp = value

Parameters: value Floating-point number to be broken into a [0.5, 1.0) value and a power of 2

exp Pointer to storage area that holds power-of-2 value

Example: #include <math.h>

float ret, value;

int \*exp;

ret=frexpf(value, exp);

Remarks: The **frexp** function breaks **value** into a [0.5, 1.0) value and a power of 2. It stores the resultant power-of-2 value in the area pointed to by **exp**.

The **frexp** function returns the return value **ret** in the range [0.5, 1.0) or as 0.0.

If **value** is 0.0, the contents of the **int** storage area pointed to by **exp** and the value of **ret** are both 0.0.

float ldexpf (float e, int f) Converting Mantissa and Exponent into Floating-Point Number

Description: Multiplies a floating-point number by a power of 2.

Header file: <math.h>

Return values: Result of e \* 2f operation

Parameters: e Floating-point number to be multiplied by a power of 2

f Power-of-2 value

Example: #include <math.h>

float ret, e;

int f;

ret=ldexpf(e, f);

float logf (float d) Natural Logarithm

Description: Calculates the natural logarithm of a floating-point number.

Header file: <math.h>

Return values: Normal: Natural logarithm of **d**

Abnormal: Domain error: Returns not-a-number.

Parameters: d Floating-point number for which natural logarithm is to be computed

Example: #include <math.h>

float d, ret;

ret=logf(d);

Error conditions: A domain error occurs if **d** is negative.

A range error occurs if **d** is 0.0.

float log10f(float d) Base-Ten Logarithm

Description: Calculates the base-ten logarithm of a floating-point number.

Header file: <math.h>

Return values: Normal: Base-ten logarithm of **d**

Abnormal: Domain error: Returns not-a-number.

Parameters: d Floating-point number for which base-ten logarithm is to be computed

Example: #include <math.h>

float d, ret;

ret=log10f(d);

Error conditions: A domain error occurs if **d** is negative.

A range error occurs if **d** is 0.0.

float modff (float a, float \*b) Breaking Floating-Point Number into Integral and Fractional Parts

Description: Breaks a floating-point number into integral and fractional parts.

Header file: <math.h>

Return values: Fractional part of **a**

Parameters: a Floating-point number to be broken into integral and fractional parts

b Pointer indicating storage area that stores integral part

Example: #include <math.h>

float a, \*b, ret;

ret=modff(a, b);

float powf (float x, float y) Power of Floating-Point Number

Description: Calculates a power of floating-point number.

Header file: <math.h>

Return values: Normal: Value of **x** raised to the power **y**

Abnormal: Domain error: Returns not-a-number.

Parameters: x Value to be raised to a power

y Power value

Example: #include <math.h>

float x, y, ret;

ret=powf(x, y);

Error conditions: A domain error occurs if **x** is 0.0 and **y** is 0.0 or less, or if **x** is negative and **y** is not an integer.

float sqrtf (float d) Square Root

Description: Calculates the positive square root of a floating-point number.

Header file: <math.h>

Return values: Normal: Positive square root of **d**

Abnormal: Domain error: Returns not-a-number.

Parameters: d Floating-point number for which positive square root is to be computed

Example: #include <math.h>

float d, ret;

ret=sqrtf(d);

Error conditions: A domain error occurs if **d** is negative.

float ceilf (float d) Rounding Up

Description: Returns the smallest integral value not less than or equal to the given floating-point number.

Header file: <math.h>

Return values: Smallest integral value not less than or equal to **d**

Parameters: d Floating-point number for which smallest integral value not less than that number is

to be computed

Example: #include <math.h>

float d, ret;

ret=ceilf(d);

Remarks: The ceil function returns the smallest integral value not less than or equal to **d**, expressed as a **double** type value. Therefore, if **d** is negative, the value after truncation of the fractional part is returned.

float fabsf (float d) Absolute Value

Description: Calculates the absolute value of a floating-point number.

Header file: <math.h>

Return values: Absolute value of **d**

Parameters: d Floating-point number for which absolute value is to be computed

Example: #include <math.h>

float d, ret;

ret=fabsf(d);

float floorf (float d) Truncation

Description: Returns the largest integral value not greater than or equal to the given floating-point number.

Header file: <math.h>

Return values: Largest integral value not greater than or equal to **d**

Parameters: d Floating-point number for which largest integral value not greater than that number

is to be computed

Example: #include <math.h>

float d, ret;

ret=floorf(d);

Remarks: The **floor** function returns the largest integral value not greater than or equal to **d**, expressed as a **double** type value. Therefore, if **d** is negative, the value after rounding-up of the fractional part is returned.

float fmodf (float x, float y) Remainder

Description: Calculates the remainder of a division of two floating-point numbers.

Header file: <math.h>

Return values: When **y** is 0.0: **x**

When **y** is not 0.0: Remainder of division of **x** by **y**

Parameters: x Dividend

y Divisor

Example: #include <math.h>

float x, y, ret;

ret=fmodf(x, y);

Remarks: In the **fmod** function, the relationship between parameters **x** and **y** and return value ret is as follows:

x = y \* i + ret (where i is an integer)

The sign of return value **ret** is the same as the sign of **x**.

If the quotient of x/y cannot be represented, the value of the result is not guaranteed.

float acoshf(float d) Hyperbolic Arc Cosine

Description: Calculates the hyperbolic arc cosine of a floating-point number.

Header file: <math.h>

Return values: Normal: Hyperbolic arc cosine of **d**

Abnormal: Domain error: Returns **NaN**.

Parameters: d Floating-point number for which hyperbolic arc cosine is to be computed

Example: #include <math.h>

float d, ret;

ret=acoshf(d);

Error conditions: A domain error occurs when **d** is smaller than 1.0.

Remarks: The **acosh** function returns the hyperbolic arc cosine in the range [0, +∞].

float asinhf(float d) Hyperbolic Arc Sine

Description: Calculates the hyperbolic arc sine of a floating-point number.

Header file: <math.h>

Return values: Hyperbolic arc sine of **d**

Parameters: d Floating-point number for which hyperbolic arc sine is to be computed

Example: #include <math.h>

float d, ret;

ret=asinhf(d);

float atanhf(float d) Hyperbolic Arc Tangent

Description: Calculates the hyperbolic arc tangent of a floating-point number.

Header file: <math.h>

Return values: Normal: Hyperbolic arc tangent of **d**

Abnormal: Domain error: Returns **HUGE\_VAL**, **HUGE\_VALF**, or **HUGE\_VALL** depending on the function.

Range error: Returns not-a-number.

Parameters: d Floating-point number for which hyperbolic arc tangent is to be computed

Example: #include <math.h>

float d, ret;

ret=atanhf(d);

Error conditions: A domain error occurs for a value of **d** not in the range [-1, +1]. A range error may occur for a value of **d** equal to -1 or 1.

float exp2f(float d) Exponential Function

Description: Calculates the value of 2 raised to the power **d**.

Header file: <math.h>

Return values: Normal: Exponential function value of 2

Abnormal: Range error: Returns 0, or returns +**HUGE\_VAL**, +**HUGE\_VALF**, or +**HUGE\_VALL** depending on the function

Parameters: d Floating-point number for which exponential function is to be computed

Example: #include <math.h>

float d, ret;

ret=exp2f(d);

Error conditions: A range error occurs if the absolute value of **d** is too large.

float expm1f(float d) Logarithm

Description: Calculates the value of natural logarithm base e raised to the power **d** and subtracts 1 from the result.

Header file: <math.h>

Return values: Normal: Value obtained by subtracting 1 from natural logarithm base **e** raised to the power **d**

Abnormal: Range error: Returns -**HUGE\_VAL**, -**HUGE\_VALF**, or -**HUGE\_VALL** depending on the function.

Parameters: d Power value to which natural logarithm base e is to be raised

Example: #include <math.h>

float d, ret;

ret=expm1f(d);

Error conditions: A range error occurs if **d** is too large.

Remarks: **expm1(d)** provides more accurate calculation than **exp(x) - 1** even when **d** is near to 0.

int ilogbf(float d) Extracting Exponent

Description: Extracts the exponent of **d**.

Header file: <math.h>

Return values: Normal: Exponential function value of **d**

d is ∞: **INT\_MAX**

d is not-a-number: **FP\_ILOGBNAN**

d is 0: **FP\_ILOGBNAN**

Abnormal: d is 0 and a range error has occurred: **FP\_ILOGB0**

Parameters: d Value of which exponent is to be extracted

Example: #include <math.h>

float d;

int ret;

ret = ilogbf(d);

Error conditions: A range error may occur if **d** is 0.

float log1pf(float d) Logarithm

Description: Calculates the natural logarithm (base e) of **d** + 1.

Header file: <math.h>

Return values: Normal: Natural logarithm of **d** + 1

Abnormal: Domain error: Returns not-a-number. Range error: Returns -**HUGE\_VAL**, -**HUGE\_VALF**, or -**HUGE\_VALL** depending on the function.

Parameters: d Value for which the natural logarithm of this parameter + 1 is to be computed

Example: #include <math.h>

float d, ret;

ret = log1pf(d);

Error conditions: A domain error occurs if **d** is smaller than -1.

A range error occurs if **d** is -1.

Remarks: **log1p(d)** provides more accurate calculation than **log(1+d)** even when **d** is near to 0.

float log2f(float d) Logarithm

Description: Calculates the base-2 logarithm of **d**.

Header file: <math.h>

Return values: Normal: Base-2 logarithm of **d**

Abnormal: Domain error: Returns not-a-number.

Parameters: d Value of which logarithm is to be calculated

Example: #include <math.h>

float d, ret;

ret = log2f(d);

Error conditions: A domain error occurs if **d** is a negative value.

float logbf(float d) Extracting Exponent

Description: Extracts the exponent of **d** in internal floating-point representation, as a floating-point value.

Header file: <math.h>

Return values: Normal: Signed exponent of **d**

Abnormal: Range error: Returns –**HUGE\_VAL**, –**HUGE\_VALF**, or –**HUGE\_VALL** depending on the function.

Parameters: d Value of which exponent is to be extracted

Example: #include <math.h>

float d, ret;

ret = logbf(d);

Error conditions: A range error may occur if **d** is 0.

Remarks: **d** is always assumed to be normalized.

float scalbnf(float d, long e)

float scalblnf(float d, long int e) Multiplication between Floating-Point Number and FLT\_RADIX

Description: Calculates a floating-point number multiplied by a power of radix, which is an integer.

Header file: <math.h>

Return values: Normal: Value equal to **d** multiplied by **FLT\_RADIX**

Abnormal: Range error: Returns -**HUGE\_VAL,** -**HUGE\_VALF**, or -**HUGE\_VALL** depending on the function.

Parameters: d Value to be multiplied by **FLT\_RADIX** raised to the power **e**

e Exponent used to compute a power of **FLT\_RADIX**

Example: #include <math.h>

float d, ret;

long e;

ret = scalbnf(d,e);

Error conditions: A range error may occur if **d** is 0.

Remarks: **FLT\_RADIX** raised to the power **e** is not actually calculated.

float cbrtf(float d) Cube Root

Description: Calculates the cube root of a floating-point number.

Header file: <math.h>

Return values: Cube root of **d**

Parameters: d Value for which a cube root is to be computed

Example: #include <math.h>

float d, ret;

ret = cbrtf(d);

float hypotf(float d, double e) Euclidean Distance

Description: Calculates the square root of the sum of floating-point numbers raised to the power 2.

Header file: <math.h>

Return values: Normal: Square root function value of sum of **d** raised to the power 2 and **e** raised to the power 2

Abnormal: Range error: Returns **HUGE\_VAL**, **HUGE\_VALF**, or **HUGE\_VALL** depending on the function.

Parameters: d Values for which the square root of the sum of these values

e raised to the power 2 is to be computed

Example: #include <math.h>

double e;

float d, ret;

ret = hypotf(d, e);

Error conditions: A range error may occur if the result overflows.

float erff(float d) Error

Description: Calculates the error function value of a floating-point number.

Header file: <math.h>

Return values: Error function value of **d**

Parameters: d Value for which the error function value is to be computed

Example: #include <math.h>

float d, ret;

ret = erff(d);

float erfcf(float d) Complementary Error

Description: Calculates the complementary error function value of a floating-point number.

Header file: <math.h>

Return values: Complementary error function value of **d**

Parameters: d Value for which the complementary error function value is to be computed

Example: #include <math.h>

float d, ret;

ret = erfcf(d);

Error conditions: A range error occurs if the absolute value of **d** is too large.

float lgammaf(float d) Logarithm of Gamma Function

Description: Calculates the logarithm of the gamma function of a floating-point number.

Header file: <math.h>

Return values: Normal: Logarithm of gamma function of **d**

Abnormal: Domain error: Returns **HUGE\_VAL**, **HUGE\_VALF**, or **HUGE\_VALL** with the mathematically correct sign.

Range error: Returns +**HUGE\_VAL**, +**HUGE\_VALF**, or +**HUGE\_VALL**.

Parameters: d Value for which the logarithm of the gamma function is to be computed

Example: #include <math.h>

float d, ret;

ret = lgammaf(d);

Error conditions: A range error is set if the absolute value of **d** is too large or small.

A domain error occurs if **d** is a negative integer or 0 and the calculation result is not representable.

float tgammaf(float d) Gamma

Description: Calculates the gamma function of a floating-point number.

Header file: <math.h>

Return values: Normal: Gamma function value of **d**

Abnormal: Domain error: Returns **HUGE\_VAL**, **HUGE\_VALF**, or **HUGE\_VALL** with the same sign as that of **d**.

Range error: Returns 0, or returns +**HUGE\_VAL**, +**HUGE\_VALF**, or +**HUGE\_VALL** with the mathematically correct sign depending on the function.

Parameters: d Value for which the gamma function value is to be computed

Example: #include <math.h>

float d, ret;

ret = tgammaf(d);

Error conditions: A range error is set if the absolute value of **d** is too large or small.

A domain error occurs if **d** is a negative integer or 0 and the calculation result is not representable.

float nearbyintf(float d) Conversion to Integer

Description: Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction.

Header file: <math.h>

Return values: **d** rounded to an integer in the floating-point format

Parameters: d Value to be rounded to an integer in the floating-point format

Example: #include <math.h>

float d, ret;

ret = nearbyintf(d);

Remarks: The **nearbyint** function group does not generate "inexact" floating-point exceptions.

float rintf(float d) Conversion to Integer

Description: Rounds a floating-point number to an integer in the floating-point representation according to the current rounding direction.

Header file: <math.h>

Return values: **d** rounded to an integer in the floating-point format

Parameters: d Value to be rounded to an integer in the floating-point format

Example: #include <math.h>

float d, ret;

ret = rintf(d);

Remarks: The **rint** function group differs from the **nearbyint** function group only in that the **ring** function group may generate "inexact" floating-point exceptions.

long int lrintf(float d) Conversion to Integer

Description: Rounds a floating-point number to the nearest integer according to the current rounding direction.

Header file: <math.h>

Return values: Normal: **d** rounded to an integer

Abnormal: Range error: Returns an undetermined value.

Parameters: d Value to be rounded to an integer

Example: #include <math.h>

float d;

long ret;

ret = lrintf(d);

Error conditions: A range error may occur if the absolute value of **d** is too large.

Remarks: The return value is unspecified when the rounded value is not in the range of the return value type.

float roundf(float d)

long int lroundf(float d) Conversion to Integer

Description: Rounds a floating-point number to the nearest integer.

Header file: <math.h>

Return values: Normal: **d** rounded to an integer

Abnormal: Range error: Returns an undetermined value.

Parameters: d Value to be rounded to an integer

Example: #include <math.h>

float d;

long int ret;

ret = lroundf(d);

Error conditions: A range error may occur if the absolute value of **d** is too large.

Remarks: When **d** is at the midpoint between two integers, the **lround** function group selects the integer farther from 0 regardless of the current rounding direction. The return value is unspecified when the rounded value is not in the range of the return value type.

float truncf(float d) Conversion to Integer

Description: Rounds a floating-point number to the nearest integer in the floating-point representation.

Header file: <math.h>

Return values: **d** truncated to an integer in the floating-point format

Parameters: d Value to be rounded to an integer in the floating-point representation

Example: #include <math.h>

float d, ret;

ret = truncf(d);

Remarks: The **trunc** function group rounds **d** so that the absolute value after rounding is not greater than the absolute value of **d**.

float remainderf(float d1, float d2) Floating-Point Remainder Calculation

Description: Calculates the remainder of a division of two floating-point numbers.

Header file: <math.h>

Return values: Remainder of division of **d1** by **d2**

Parameters: d1 Values for which remainder of a division is to be computed

d2

Example: #include <math.h>

float d1, d2, ret;

ret = remainderf(d1, d2);

Remarks: The remainder calculation by the **remainder** function group conforms to the IEEE 60559 standard.

float remquof(float d1, float d2, long \*q) Floating-Point Remainder Calculation

Description: Calculates the remainder of a division of two floating-point numbers.

Header file: <math.h>

Return values: Remainder of division of **d1** by **d2**

Parameters: d1 Values for which remainder of a division is to be computed

d2

q Value pointing to the location to store the quotient obtained by remainder calculation

Example: #include <math.h>

float d1, d2, ret;

long q;

ret = remquof(d1, d2, &q);

Remarks: The value stored in the location indicated by **q** has the same sign as the result of **x/y** and the integral quotient of modulo-2n **x/y** (**n** is an implementation-defined integer equal to or greater than 3).

float copysignf(float d1, float d2) Sign Copy

Description: Generates a value consisting of the absolute value of **d1** and the sign of **d2**.

Header file: <math.h>

Return values: Normal: Value consisting of absolute value of **d1** and sign of **d2**

Abnormal: Range error: Returns an undetermined value.

Parameters: d1 Value of which absolute value is to be used in the generated value

d2 Value of which sign is to be used in the generated value

Example: #include <math.h>

float d1, d2, ret;

ret = copysignf(d1, d2);

Remarks: When **d1** is a not-a-number, the **copysign** function group generates a not-a-number with the sign bit of **d2**.

float nanf(const char \*c) Not-a-Number

Description: Returns not-a-number.

Header file: <math.h>

Return values: **qNaN** with the contents of the location indicated by c or 0 (when **qNaN** is not supported)

Parameters: c Pointer to a string

Example: #include <math.h>

float ret;

const char \*c;

ret = nanf(c);

Remarks: The **nan("c string")** call is equivalent to **strtod("NAN(c string)", (char\*\*) NULL)**. The **nanf** and **nanl** calls are equivalent to the corresponding **strtof** and **strtold** calls, respectively.

float nextafterf(float d1, float d2) Floating-Point Manipulation

Description: Calculates the next floating-point representation following **d1** in the direction to **d2** on the real axis.

Header file: <math.h>

Return values: Normal: Representable floating-point value

Abnormal: Range error: Returns **HUGE\_VAL**, **HUGE\_VALF**, or **HUGE\_VALL** with the mathematically correct sign depending on the function.

Parameters: d1 Floating-point value on the real axis

d2 Value indicating the direction viewed from **d1**, in which a representable floating-point value is to be found

Example: #include <math.h>

float d1, d2, ret;

ret = nextafterf(d1, d2);

Error conditions: A range error may occur if **d1** is the maximum finite value that can be represented in its type and the return value is an infinity or cannot be represented in its type.

Remarks: The **nextafter** function group returns **d2** when **d1** is equal to **d2**.

float fdimf(float d1, float d2) Positive Difference

Description: Calculates the positive difference between two arguments.

Header file: <math.h>

Return values: Normal: Positive difference between two arguments

Abnormal: Range error: **HUGE\_VAL**, **HUGE\_VALF**, or **HUGE\_VALL**

Parameters: d1 Values of which difference is to be computed

d2

Example: #include <math.h>

float d1, d2, ret;

ret = fdimf(d1, d2);

Error conditions: A range error may occur if the return value overflows.

float fmaxf(float d1, float d2) Maximum Value

Description: Obtains the greater of two arguments.

Header file: <math.h>

Return values: Greater of two arguments

Parameters: d1 Values to be compared

d2

Example: #include <math.h>

float d1, d2, ret;

ret = fmaxf(d1, d2);

Remarks: The **fmax** function group recognizes a not-a-number as a lack of data. When one argument is a not-a-number and the other is a numeric value, the function returns the numeric value.

float fminf(float d1, float d2) Minimum Value

Description: Obtains the smaller of two arguments.

Header file: <math.h>

Return values: Smaller of two arguments

Parameters: d1 Values to be compared

d2

Example: #include <math.h>

float d1, d2, ret;

ret = fminf(d1, d2);

Remarks: The **fmin** function group recognizes a not-a-number as a lack of data. When one argument is a not-a-number and the other is a numeric value, the function returns the numeric value.

float fmaf(float d1, float d2, float d3) Multiply and Add

Description: Calculates **(d1 \* d2) + d3** as a single ternary operation.

Header file: <math.h>

Return values: Result of **(d1 \* d2) + d3** calculated as ternary operation

Parameters: d1, d2, d3 Floating-point values

Example: #include <math.h>

float d1, d2, ret;

ret = fmaf(d1, d2);

Remarks: The **fma** function group performs calculation as if infinite precision is available and rounds the result only one time in the rounding mode indicated by **FLT\_ROUNDS**.

#### <stdbool.h>

This header file defines macros relating to logical types and values.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Macro  (variable) | bool | \_Bool | Define **\_Bool** variable. |
| Macro  (constant) | true | 1 | Define True as 1. |
| false | 0 | Define False as 0. |
| \_\_bool\_true\_false\_are\_defined | 1 | Define Boolean for True and False |

#### <stddef.h>

Defines macro names used in common in the standard include files.

The following macro names are all implementation-defined.

|  |  |  |
| --- | --- | --- |
| Type | Definition Name | Description |
| Type (macro) | ptrdiff\_t | Indicates the type of the result of subtraction between two pointers. |
| size\_t | Indicates the type of the result of an operation using the **sizeof** operator. |
| Constant (macro) | NULL | Indicates the value when a pointer is not pointing at anything.  This value is such that the result of a comparison with 0 using the equality operator (==) is true. |
| Function (macro) | offsetof | Obtains the offset in bytes from the beginning of a structure to a structure member. |

#### <stdint.h>

This header file defines macros only.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Macro | int\_least8\_t  uint\_least8\_t  int\_least16\_t  uint\_least16\_t  int\_least32\_t  uint\_least32\_t  int\_least64\_t  uint\_least64\_t |  | Indicates the types whose size is large enough to store signed and unsigned integer types of 8, 16, 32, and 64 bits. |
| int\_fast8\_t  uint\_fast8\_t  int\_fast16\_t  uint\_fast16\_t  int\_fast32\_t  uint\_fast32\_t  int\_fast64\_t  uint\_fast64\_t |  | Indicates the types which can operate signed and unsigned integer types of 8, 16, 32, and 64 bits at the fastest speed. |
| intptr\_t  uintptr\_t |  | These indicate signed and unsigned integer types that can be converted to or from pointers to **void**. |
| intmax\_t  uintmax\_t |  | These indicate signed and unsigned integer types that can represent all signed and unsigned integer types. |
| Constant (macro) | INTPTR\_MIN  INTPTR\_MAX  UINTPTR\_MAX | -2147483648  2147483647  4294967295 | Indicates the minimum value of pointer-holding signed integer type.  Indicates the maximum value of pointer-holding signed integer type.  Indicates the maximum value of pointer-holding unsigned integer type. |
| INTMAX\_MIN  INTMAX\_MAX  UINTMAX\_MAX | -2147483648  2147483647  4294967295 | Indicates the minimum value of greatest-width signed integer type.  Indicates the maximum value of greatest-width signed integer type.  Indicates the maximum value of greatest-width unsigned integer type. |
| PTRDIFF\_MIN  PTRDIFF\_MAX | -2147483648  2147483647 | Indicates the minimum/maximum value representable as type ptrdiff\_t. Note that the definition shown here is merely representative |
| SIZE\_MAX | 4294967295 | Indicates the maximum value representable as type size\_t. Note that the definition shown here is merely representative |

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Constant (macro) | INT8\_MIN | -128 | Indicates the minimum value representable as type int8\_t |
| INT16\_MIN | -32768 | Indicates the minimum value representable as type int16\_t |
| INT32\_MIN | -2147483648 | Indicates the minimum value representable as type int32\_t |
| INT8\_MAX | 127 | Indicates the maximum value representable as type int8\_t |
| INT16\_MAX | 32767 | Indicates the maximum value representable as type int16\_t |
| INT32\_MAX | 2147483647 | Indicates the maximum value representable as type int16\_t |
| UINT8\_MAX | 255 | Indicates the maximum value representable as type uint8\_t |
| UINT16\_MAX | 65535 | Indicates the maximum value representable as type uint16\_t |
| UINT32\_MAX | 4294967295 | Indicates the maximum value representable as type uint32\_t |
| INT\_LEAST8\_MIN | -128 | Indicates the minimum value representable as type int\_least8\_t |
| INT\_LEAST16\_MIN | -32768 | Indicates the minimum value representable as type int\_least16\_t |
| INT\_LEAST32\_MIN | -2147483648 | Indicates the minimum value representable as type int\_least32\_t |
| INT\_LEAST8\_MAX | 127 | Indicates the maximum value representable as type int\_least8\_t |
| INT\_LEAST16\_MAX | 32767 | Indicates the maximum value representable as type int\_least16\_t |
| INT\_LEAST32\_MAX | 2147483647 | Indicates the maximum value representable as type int\_least32\_t |
| UINT\_LEAST8\_MAX | 255 | Indicates the maximum value representable as type uint\_least8\_t |
| UINT\_LEAST16\_MAX | 65535 | Indicates the maximum value representable as type uint\_least16\_t |
| UINT\_LEAST32\_MAX | 4294967295 | Indicates the maximum value representable as type uint\_least32\_t |
| INT\_FAST8\_MIN | -128 | Indicates the minimum value representable as type int\_fast8\_t |
| INT\_FAST16\_MIN | -32768 | Indicates the minimum value representable as type int\_fast16\_t |
| INT\_FAST32\_MIN | -2147483648 | Indicates the minimum value representable as type int\_fast32\_t |
| INT\_FAST8\_MAX | 127 | Indicates the maximum value representable as type int\_fast8\_t |

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Constant (macro) | INT\_FAST16\_MAX | 32767 | Indicates the maximum value representable as type int\_fast16\_t |
| INT\_FAST32\_MAX | 2147483647 | Indicates the maximum value representable as type int\_fast32\_t |
| UINT\_FAST8\_MAX | 255 | Indicates the maximum value representable as type uint\_fast8\_t |
| UINT\_FAST16\_MAX | 65535 | Indicates the maximum value representable as type uint\_fast16\_t |
| UINT\_FAST32\_MAX | 4294967295 | Indicates the maximum value representable as type uint\_fast32\_t |

#### <stdlib.h>

Defines standard functions for standard processing of C programs.

The following macros are implementation-defined.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Constant (macro) | RAND\_MAX | 1073741823 | Indicates the maximum value of pseudo-random integers generated by the **rand** function. |
| EXIT\_FAILURE | 1 | Indicates the value of the status argument to exit that reports unsuccessful termination |
| EXIT\_SUCCESS | 0 | Indicates the successfully completed state. |
| Function | rand |  | Generates pseudo-random integers from 0 to **RAND\_MAX**. |
| srand |  | Sets an initial value of the pseudo-random number sequence generated by the **rand** function. |
| abs |  | Calculates the absolute value of an **int** type integer. |
| labs |  | Calculates the absolute value of a **long** type integer. |

int rand (void) Pseudo-Random Number Generation

Description: Generates a pseudo-random integer from 0 to **RAND\_MAX**.

Header file: <stdlib.h>

Return values: Pseudo-random integer

Example: #include <stdlib.h>

int ret;

ret=rand();

void srand (unsigned int seed) Initial Setting for Pseudo-Random Number Sequence

Description: Sets an initial value of the pseudo-random number sequence generated by the **rand** function.

Header file: <stdlib.h>

Parameters: seed Initial value for pseudo-random number sequence generation

Example: #include <stdlib.h>

unsigned int seed;

srand(seed);

Remarks: The **srand** function sets up an initial value for pseudo-random number sequence generation of the **rand** function. If pseudo-random number sequence generation by the **rand** function is repeated and if the same initial value is set up again by the **srand** function, the same pseudo-random number sequence is repeated.

If the **rand** function is called before the **srand** function, 1 is set as the initial value for the pseudo-random number generation.

int abs (int i) Absolute Value

Description: Calculates the absolute value of an **int** type integer.

Header file: <stdlib.h>

Return values: Absolute value of i

Parameters: i Integer to calculate the absolute value of

Example: #include <stdlib.h>

int i, ret;

ret=abs(i);

Remarks: If the resultant absolute value cannot be expressed as an **int** type integer, correct operation is not guaranteed.

long labs (long j) Absolute Value

Description: Calculates the absolute value of a **long** type integer.

Header file: <stdlib.h>

Return values: Absolute value of j

Parameters: j Integer to calculate the absolute value of

Example: #include <stdlib.h>

long j;

long ret;

ret=labs(j);

Remarks: If the resultant absolute value cannot be expressed as a **long** type integer, correct operation is not guaranteed.

#### mem<string.h>

Defines functions for handling character arrays.

|  |  |  |
| --- | --- | --- |
| Type | Definition Name | Description |
| Function | memcpy | Copies contents of a source storage area of a specified length to a destination storage area. |
| memcmp | Compares two storage areas specified. |
| memset | Sets a specified character for a specified number of times at the beginning of a specified storage area. |
| memmove | Copies contents of a source storage area of a specified length to a destination storage area. Even if a part of the source storage area and a part of the destination storage area overlap, correct copy is performed. |

When using functions defined in this standard include file, note the following.

(1) On copying a string, if the destination area is smaller than the source area, correct operation is not guaranteed.

Example

char a[]="abc";

char b[3];

.

.

.

strcpy (b, a);

In the above example, the size of array **a** (including the null character) is 4 bytes. Copying by strcpy overwrites data beyond the boundary of array **b**.



(2) On copying a string, if the source area overlaps the destination area, correct operation is not guaranteed.

Example

int a[ ]="a";

:

:

strcpy(&a[1], a);

:

In the above example, before the null character of the source is read, 'a' is written over the null character. Then the subsequent data after the source string is overwritten in succession.



void \*memcpy (void \*s1, const void \*s2, size\_t n) Storage Area Copy

Description: Copies the contents of a source storage area of a specified length to a destination storage area.

Header file: <string.h>

Return values: **s1** value

Parameters: s1 Pointer to destination storage area

s2 Pointer to source storage area

n Number of characters to be copied

Example: #include <string.h>

void \*ret, \*s1;

const void \*s2;

size\_t n;

ret=memcpy(s1, s2, n);

int memcmp (const void \*s1, const void \*s2, size\_t n) Storage Area Comparison

Description: Compares the contents of two storage areas specified.

Header file: <string.h>

Return values: If storage area pointed by s1 > storage area pointed by **s2**: Positive value

If storage area pointed by s1 == storage area pointed by **s2**: 0

If storage area pointed by s1 < storage area pointed by **s2**: Negative value

Parameters: s1 Pointer to the reference storage area to be compared

s2 Pointer to the storage area to compare to the reference

n Number of characters to compare

Example: #include <string.h>

const void \*s1, \*s2;

size\_t n;

int ret;

ret=memcmp(s1, s2, n);

Remarks: The **memcmp** function compares the contents of the first **n** characters in the storage areas pointed by **s1** and **s2**. The rules of comparison are implementation-defined.

void \*memset (void \*s, int c, size\_t n) Character Repeating

Description: Sets a specified character a specified number of times at the beginning of a specified storage area.

Header file: <string.h>

Return values: Value of s

Parameters: s Pointer to storage area to set characters in

c Character to be set

n Number of characters to be set

Example: #include <string.h>

void \*s, \*ret;

int c;

size\_t n;

ret=memset(s, c, n);

Remarks: The **memset** function sets the character specified by **c** a number of times specified by **n** in the storage area specified by **s**.

void \*memmove (void \*s1, const void \*s2, size\_t n) Storage Area Move

Description: Copies the specified size of the contents of a source area to a destination storage area. If part of the source storage area and the destination storage area overlap, data is copied to the destination storage area before the overlapped source storage area is overwritten. Therefore, correct copy is enabled.

Header file: <string.h>

Return values: Value of s1

Parameters: s1 Pointer to the destination storage area

s2 Pointer to the source storage area

n Number of characters to be copied

Example: #include <string.h>

void \*ret, \*s1;

const void \*s2;

size\_t n;

ret=memmove(s1, s2, n);

#### <iso646.h>

Define constant macros conforming to ISO 646 standard.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Definition Name | Definition Value | Description |
| Macro | and | && | Logical AND operator. |
| and\_eq | &= | Bitwise AND assignment operator. |
| bitand | & | Binary AND operator. |
| bitor | | | Binary OR operator. |
| compl | ~ | Binary Ones Complement operator. |
| not | ! | Logical NOT operator. |
| not\_eq | != | Logical not equal to operator. |
| or | || | Logical OR Operator. |
| or\_eq | |= | Bitwise inclusive OR and assignment operator. |
| xor | ^ | Binary XOR operator. |
| xor\_eq | ^= | Bitwise exclusive OR and assignment operator. |