RX-family C/C++ Compiler Package

Application Notes: Compiler Usage Guide Languages Edition (C89, C99)

REJ06J0100-0100
Rev.1.00
Apr 20, 2010

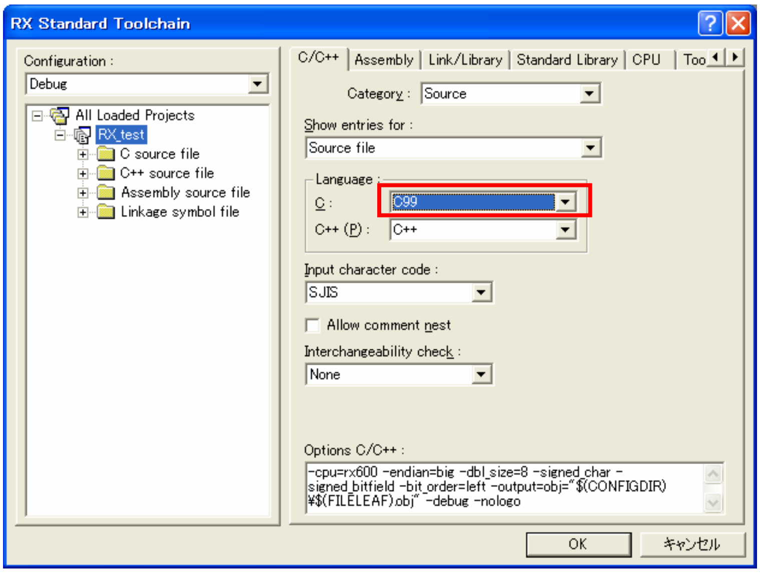
This document introduces the C99 language specifications added to version 1.0 of the RX-family C/C++ compiler.

Table of contents

1. Introduction

Version 1.0 of the RX-family C/C++ compiler works not only with the ANSI-standard C language C89 and the ANSI-standard C++ language and EC++ language, but also with the successor to C89, the ANSI-standard C language C99 (except for variable-length arrays). This document explains the new functionality in C99, and the precautions necessary when migrating from C89 to C99.

The RX-family C/C++ compiler compiles source programs for C89 by default (the -lang=c compile option). To compile for C99 from the command line, specify the -lang=c99 compile option. This option can be set in the RX Standard Toolchain dialog box displayed by choosing the Build and then RX Standard Toolchain menus in High-performance Embedded Workshop.



Configuration : Debug

Category : Source

Show entries for : Source file

Language :

C : C99

C++ (P) : C++

Input character code : SJIS

☐ Allow comment nest

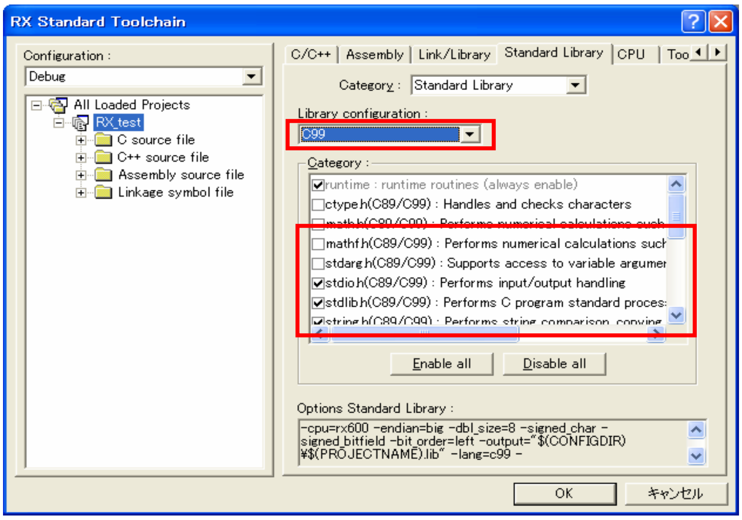
Interchangeability check : None

Options C/C++ :

```
-cpu=rx600 -endian=big -dbl_size=8 -signed_char -
signed_bitfield -bit_order=left -output=obj="$(CONFIGDIR)
¥$(FILELEAF).obj" -debug -nologo
```

Figure 1-1 Compiler options for C99

Some of the standard libraries added to C99 require setup using the standard library configuration tool. For the command line, use the -lang=c99 and -head options, which can be set in the RX Standard Toolchain dialog box in High-performance Embedded Workshop. For details, see *2.5 Standard include files*.



Configuration : Debug

Category : Standard Library

Library configuration : C99

Category :

- ☑ runtime : runtime routines (always enable)
- ☐ ctype.h(C89/C99) : Handles and checks characters
- ☐ math.h(C89/C99) : Performs numerical calculations such
- ☐ mathf.h(C89/C99) : Performs numerical calculations such
- ☐ stdarg.h(C89/C99) : Supports access to variable argumen
- ☑ stdio.h(C89/C99) : Performs input/output handling
- ☑ stdlib.h(C89/C99) : Performs C program standard proces
- ☑ string.h(C89/C99) : Performs string comparison, copying

Options Standard Library :

```
-cpu=rx600 -endian=big -dbl_size=8 -signed_char -
signed_bitfield -bit_order=left -output="$(CONFIGDIR)
¥$(PROJECTNAME).lib" -lang=c99 -
```

Figure 1-2 Standard library configuration tool option for C99

2. New functionality in C99

2.1 Primitive types

The primitive types added to C99 include logical types, complex number types, long long types, variable-length array types, and flexible array members.

2.1.1 Logical types

■ Overview

Logical types indicating either true or false have been added.

■ Description

The bool type and _Bool type have been added to C99 as logical types indicating true or false. The values that can be used for the bool type (_Bool type) are true, indicating truth, and false, indicating falsity. To use the bool type, _Bool type, true, and false, include stdbool.h.

■ Example usage

(1) Using the bool types true and false

The following example uses a bool type variable.

```
#include <stdbool.h>

int func(bool b, bool c)
{
   if (b == true) {
      return 1;
   }

   if (c == false) {
      return -1;
   }

   return 0;
}
```

(2) Conditional expressions

The following example returns the results of comparing int type variables as a bool type.

```
#include <stdbool.h>

bool func(int a, int b)
{
   return (a==b);
}
```

2.1.2 Complex number types

■ Overview

The _Complex type has been added to handle complex numbers, and the _Imaginary type has been added to represent the imaginary component of a complex number.

■ Description

Types have been added to C99 to handle complex numbers. Complex numbers consist of a real component and an imaginary component. The three types that can be used as complex number types are the float _Complex type, double _Complex type, and long double _Complex type. For example, the float _Complex type represents a complex number type where both the real component and imaginary component are a float type. Also, the three types that can be used as the imaginary component of a complex number are the float _Imaginary type, double _Imaginary type, and long double _Imaginary type. For example, the float _Imaginary type means that the float type is imaginary. __I__ can also be used as the imaginary unit (defined mathematically as *i*, where *i*^2=-1). Note that "complex" can be used in place of "_Complex", "imaginary" can be used in place of "_Imaginary", and "I" can be used in place of "__I__". To use complex number types, include complex.h.

The -head=complex option must be specified for the standard library configuration tool in order to use the complex number calculation library.

■ Example usage

(1) Using complex number types

```
#include <complex.h>

float _Complex cf;
double _Complex cd;
long double _Complex cld;

void func()
{
   /* A float type complex number with a real component of 1.0 and an imaginary component of 2.0
*/
   cf = 1.0f + __I__ * 2.0f;

   /* A double type complex number with a real component of 10.0 and an imaginary component of 20.0*/
   cd = 10.0 + __I__ * 20.0;

   /* A long double type complex number with a real component of 100.0 and an imaginary component
of 200.0 */
   cld = 100.0 + __I__ * 200.0;
}
```

(2) Calculating complex number types

```
#include <complex.h>

float _Complex cf1, cf2, cf3;

void func1()
{
   /* Adding complex numbers */
   cf1 = cf2 + cf3;
}

void func2()
{
   /* Subtracting complex numbers */
   cf1 = cf2 - cf3;
}

void func3()
{
   /* Multiplying complex numbers */
   cf1 = cf2 * cf3;
}

void func4()
{
   /* Dividing complex numbers */
   cf1 = cf2 / cf3;
}

int func5()
{
   /* Comparing complex numbers */
   if (cf1 == cf2) return 1;
   else if (cf1 != cf2) return -1;
}
```

(3) Converting types

```
#include <complex.h>

float _Complex cf;
double _Complex cd;
float fr;
float _Imaginary fi;

void func1()
{
   /* Converting a float_Complex to a float type to obtain the real component */
   fr = (float)cf;
}

void func2()
{
   /* Converting a float_Complex to a float Imaginary type to obtain the imaginary component */
   fi = (float _Imaginary)cf;
}

void func3()
{
   /* Converting a float to a float_Complex type, where the imaginary component for cf is 0  */
   cf = (float _Complex)fr;
}

void func4()
{
   /* Converting a float Imaginary to a float Complex type, where the real component for cf is 0 */
   cf = (float _Complex)fi;
}

void func5()
{
   /* Converting a float_Complex to a double Complex type.
   ** Both the real component and imaginary component are converted from a float to a double type.
   */
   df = (double _Complex)cf;
}
```

(4) Imaginary units

```
#include <complex.h>

float _Complex cf1, cf2;

Void func()
{
   /* Replacing the cf2 real component with the cf1 imaginary component, and the cf2 (imaginary component
x -1) with the cf1 real component */
   cf1 = cf2 * __I__;

   /* The real component is -1, and the imaginary component is 0. */
   cf2 = __I__ * __I__;
}
```

2.1.3 long long types

■ Overview

Two long long types have been added to handle 64-bit integer types.

■ Description

The long long types (signed long long type and unsigned long long type) have been officially added to the C99 language specification to handle 64-bit integer types.

The long long types can also be used by the RX-family C/C++ compiler for C89.

The following gives the value ranges that the long long types can handle.

Table 2-1 Value ranges for the long long types

Type	Minimum value	Maximum value
signed long long type	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long type	0	18,446,744,073,709,551,615

The following suffixes have been added to represent long long type integer constants.

Table 2-2 Constant suffixes for the long long type

Integer constant type	Suffix
signed long long type	LL, ll
unsigned long long type	ULL, LLU, ull, llu

Formatting such as %llx, %llX, %lld, and %llu has been added for printf and sprintf to handle long long types.

Table 2-3 Formatting for the long long types

Type	Format
signed long long type	%lld (Changes the format of a value of the signed long long type to decimal) %llx (Changes the format of a value of the signed long long type to hexadecimal)
unsigned long long type	%llu (Changes the format of a value of an unsigned long long type to decimal) %llX (Changes the format of a value of an unsigned long long type to hexadecimal)

■ Example usage

The following example uses the long long types.

```
#include <stdio.h>    /* for printf */

long long sll;
unsigned long long ull;

void func1()
{
    sll = 0x7FFFFFFFFFFFFFFFLL;    /* Integer constant of the signed long long type */
    ull = 0xFFFFFFFFFFFFFFFFULL;   /* Integer constant of the unsigned long long type */
}

void func2()
{
    printf("%lld¥n", sll);    /* Outputs the value of a signed long long type variable in decimal */
    printf("%llx¥n", sll);    /* Outputs the value of a signed long long type variable in hexadecimal */

    printf("%llu¥n", ull);    /* Outputs the value of a unsigned long long type variable in decimal */
    printf("%llX¥n", ull);    /* Outputs the value of a unsigned long long type variable in hexadecimal */
}
```

2.1.4 Variable-length array types

■ Overview

Variable-length array types are now available, but the RX-family C/C++ compiler does not support them.

2.1.5 Flexible array members

■ Overview

Array members with no element count specified (flexible array members) can now be declared as the last member of a structure or union.

■ Description

In C89, the element count must be specified when an array type structure or union member is declared. In C99, an array member with no element count specified (flexible array member) can now be declared as the last member of a structure or union.

For structures that handle image data formats like bitmap or transmission data formats like TCP/IP, header information such as the size of the structure data is sometimes declared at the start of the structure, with the data after the header information declared as an array type. Also, the size of the data is sometimes not known at execution. Flexible array members can be used to facilitate handling for these kinds of data formats.

■ Example usage

In the following example, the member data for structure S is a flexible array member. For C89, the dummy data[1] needs to be declared in structure S. This means that when data is secured for structure S, the size of data[1] is subtracted from the size calculated using sizeof(struct S).

C89

```
#include <stdlib.h>    /* for malloc */

struct S {
   int size;
   int dummy1;
   int dummy2;
   int data[1];
};

void func(int size)
{
   struct S *p;
   int i;

   p = (struct S*)malloc(
            sizeof(struct S)
            - sizeof(int)
            + sizeof(int) * size);

   p->size = size;
   for (i=0; i<size; i++) {
      p->data[i] = 0;
   }
   ...
}
```

C99

```
#include <stdlib.h>    /* for malloc */

struct S {
   int size;
   int dummy1;
   int dummy2;
   int data[];   /* Flexible array member */
};

void func(int size)
{
   struct S *p;
   int i;

   p = (struct S*)malloc(
            sizeof(struct S)

            + sizeof(int) * size);

   p->size = size;
   for (i=0; i<size; i++) {
      p->data[i] = 0;
   }
   ...
}
```

■ Notes

(1) Using the sizeof operator

The return value for the sizeof operator invoked on a structure or union containing a flexible array member does not contain the size of flexible array members. In the following example, 12, the structure size not including data[], is stored for `x`.

```
void func()
{
    int x = sizeof(struct S);
}
```

(2) Variables for structure and union types that include flexible array members

When a variable is declared for a structure or union type containing a flexible array member (does not include cases for pointer types for structures or unions containing flexible array members), access to the flexible array members for that variable is not guaranteed. In the following example, since the structure S type variable `s` only has the size, dummy1, and dummy2 areas, access to s.data[0] is invalid.

```
struct S s;
void func()
{
    s.data[0] = 0;
}
```

(3) Copying structures and unions containing flexible array members

When a structure or union containing a flexible array member is copied, the flexible array members are not copied. In the following example, a copy from `*p` to `*q` results in size, dummy1, and dummy2 being copied, but not the area for data[].

```
void func(struct S *p, struct S *q)
{
    *q = *p;
}
```

2.2 Keywords

The keywords inline, restrict, and _Pragma have been added to C99.

2.2.1 inline

■ Overview

The inline keyword is now supported to instruct the compiler to perform inline expansion for functions.

■ Description

When the inline keyword is specified for a function, the compiler is instructed to perform inline expansion of the function. However, note that the inline keyword is only a hint that the compiler should perform inline expansion, and whether inline expansion is actually performed depends on the compiler processing.

■ Example usage

The following uses inline to instruct the compiler to perform inline expansion for the add function.

Source code

```
static inline int add(int x, int y)
{
    return (x + y);
}

int xxx, yyy, zzz;
void func()
{
    xxx = add(yyy, zzz);
    yyy = add(yyy, 2);
}
```

Compiler interpretation (when the -inline=100 option is specified)

```
int xxx, yyy, zzz;
void func()
{
    xxx = yyy + zzz;
    yyy = yyy + 2;
}
```

■ Notes

When inline is specified in a declaration for a function that has external linkage (function for which extern is also declared), the function needs to be defined within the same source file. If no function definition exists, an undefined symbol error (L2310) may be output during linking. Make sure that functions are defined in the same source file when a function is specified with inline.

```
inline int add(int x, int y);   /* No definition, only declaration */

int xxx, yyy, zzz;
void func()
{
    xxx = add(yyy, zzz);
}
```

Even when a function is specified as inline, inline expansion might not be performed due to compiler conditions. In this case, if extern is not specified for a function for which inline is specified, no function definition is generated. This means that an undefined symbol error (L2310) could be output during linking. When L2310 is output for a function for which inline is specified, make sure that an extern declaration is added for the corresponding function. For example, in the following example, when the -inline=0 compile option is specified, since the function add is not expanded inline, and no definition is generated for add, L2310 is output during linking. To avoid linking errors, use the extern declaration for the add function.

Before modification

```
inline int add(int x, int y)
{
   return (x + y);
}

int xxx, yyy, zzz;
void func()
{
   xxx = add(yyy, zzz);
}
```

After modification

```
inline extern int add(int x, int y)
{
   return (x + y);
}

int xxx, yyy, zzz;
void func()
{
   xxx = add(yyy, zzz);
}
```

Note that the RX-family C/C++ compiler allows the use of inline expansion, using the #pragma extension functionality (#pragma inline). When the C99 inline keyword is used, no definition is generated for functions for which inline is specified when extern is not declared, as shown above. However, when #pragma inline is used, function definitions are generated regardless of whether inline expansion is possible.

Functions for which inline is specified cannot contain references to identifiers with internal linkage (static variables within the function or static variables within the file).

```
inline void add1()
{
   /* This causes a C6030 error. */
   static int x = 0;
   x ++;
}

static int y;
inline void add2()
{
   /* This causes a C6031 error. */
   x ++;
}

void func()
{
   add1();
   add2();
}
```

2.2.2 restrict

■ Overview

The restrict keyword has been added to provide hints to the compiler regarding pointer optimizations.

■ Description

Compilers can more easily implement pointer-oriented optimizations when areas indicated by pointers modified using restrict are explicitly deemed unique from areas indicated by other pointers.

■ Example usage

(1) When restrict is not used

Usually, a compiler will assume that the areas for `*q` and `*r` may overlap with the area for `*p`. Consequently, since storing something in `*p` could cause `*q` and `*r` to be overwritten, the generated code must load `*q` and `*r` for each iteration within a loop.

Source program:

```
void func(int * p,
          int * q, int * r,
          int n)
{
   int i;

   for (i=0; i<n; i++) {
      p[i] = *q + *r;
   }
}




int a[10], b, c;
void main()
{
   func(a, &b, &c, 10);
}
```

Generated code:

```
_func:
   MOV.L      R4,R15
   MOV.L      #00000000H,R4
   BRA        L11
L12:
   MOV.L      [R2],R5    ; Load (*q)
   ADD        #01H,R4
   ADD        [R3],R5    ; Load (*r)
   MOV.L      R5,[R1+]   ; Store in (*p)
L11:
   CMP        R15,R4
   BLT        L12
L13:
   RTS
```

(2) When restrict is used

The restrict modifier can be added to `q` and `r` to explicitly tell the compiler that the areas for `*q` and `*r` will not overlap with the area for `*p`. This means that the compiler can assume that `*q` and `*r` will not be overwritten even when a value is stored in `*p`, and generate optimized code in which `*q` and `*r` are only loaded once at the start of the loop.

Source program:

```
void func(int * p,
          int * restrict q, int * restrict r,
          int n)
{
   int i;

   for (i=0; i<n; i++) {
      p[i] = *q + *r;
   }
}



int a[10], b, c;
void main()
{
   func(a, &b, &c, 10);
}
```

Generated code:

```
_func:
   MOV.L      R4,R15
   MOV.L      [R2],R4    ; Load (*q)
   MOV.L      #00000000H,R5
   ADD        [R3],R4    ; Load (*r)
   BRA        L11
L12:
   MOV.L      R4,[R1+]   ; Store in (*p)
   ADD        #01H,R5
L11:
   CMP        R15,R5
   BLT        L12
L13:
   RTS
```

2.2.3 _Pragma

■ Overview

The _Pragma keyword has been added as an operator to perform the same functionality as #pragma.

■ Description

Since how #pragma is used differs depending on the processing type, how #pragma is written needs to be changed based on the specification of the compiler used. When #pragma is used often in a source program, because #pragma code needs to be disambiguated based on the compiler type every time the code appears, the readability of the source program often suffers. The _Pragma keyword added to C99 makes it easier to specify #pragma.

■ Example usage

(1) Switching optimization policies

Assuming a #pragma exists to switch the optimization policy for each function by declaration anywhere in an individual file, the following example enables optimization for the function func_001, but not the function func_002. In C89, each compiler type judgment and #pragma code needs to be included before the definitions for func_001 and func_002, but in C99, _Pragma can be used to simplify the code.

C89

```
#ifdef __RX
/* Declaration for RX compilers (with
optimization) */
#pragma option optimize=2
#else
/* Declaration for other compilers (without
optimization) */
#pragma ...
#endif





int x;

void func_001()
{
   x ++;
   x ++;
}

#ifdef __RX
/* Declaration for RX compilers (without
optimization) */
#pragma option optimize=0
#else
/* Declaration for other compilers (without
optimization) */
#pragma ...
#endif

void func_002()
{
   x ++;
   x ++;
}
```

C99

```
#ifdef __RX
/* Declaration for RX compilers */
#define OPTIMIZE_ON  _Pragma("option
optimize=2")
#define OPTIMIZE_OFF _Pragma("option
optimize=0")
#else
/* Declaration for other compilers */
#define OPTIMIZE_ON  _Pragma("...")
#define OPTIMIZE_OFF _Pragma("...")
#endif



int x;

OPTIMIZE_ON void func_001()
{
   x ++;
   x ++;
}











OPTIMIZE_OFF void func_002()
{
   x ++;
   x ++;
}
```

(2) Switching endianness

Assuming a #pragma exists to switch the endianness for each variable by declaration anywhere in an individual file, the following example sets the variable x_little to little-endian, and the variable x_big to big-endian. In C89, each compiler type judgment and #pragma code needs to be included before the definitions for x_little and x_big, but in C99, _Pragma can be used to simplify the code.

C89

```
#ifdef __RX
/* Declaration for RX compilers (little endian)
*/
#pragma endian little
#else
/* Declaration for other compilers (little endian)
*/
#pragma ...
#endif



int x_little;

#ifdef __RX
/* Declaration for RX compilers (big endian) */
#pragma endian big
#else
/* Declaration for other compilers (big endian)
*/
#pragma ...
#endif

int x_big;
```

C99

```
#ifdef __RX
/* Declaration for RX compilers  */
#define ENDIAN_LITTLE   _Pragma("endian
little")
#define ENDIAN_BIG      _Pragma("endian big")
#else
/* Declaration for other compilers  */
#define ENDIAN_LITTLE   _Pragma("...")
#define ENDIAN_BIG      _Pragma("...")
#endif


ENDIAN_LITTLE int x_little;











ENDIAN_BIG int x_big;
```

2.3 Literals

The following explains the literals that can now be used with C99.

2.3.1 Floating-point hexadecimal notation

■ Overview

Floating-point constant values can now be represented in hexadecimal.

■ Description

C99 allows floating-point numbers to be expressed in hexadecimal to provide a smaller margin of error than expression in decimal. For example, when 0.1 is represented as a floating-point number, since it cannot be properly expressed in hexadecimal, the compiler might change the value due to rounding or margin of error for representation as hexadecimal. However, when a floating-point number is expressed in hexadecimal, the value can be fixed without being affected by rounding or margin of error. The format for expressing floating-point hexadecimals is as follows:

0xaaaa.bbbbPdd (where P is case-insensitive)

The format starts with "0x", and then has the integer component aaaa, a decimal point (.), and the decimal component bbbb. The characters following P indicate a positive or negative exponent in decimal. Note that the decimal point and decimal component can be omitted, but the characters following P cannot.

■ Example usage

When the floating-point number 0.1 is specified in decimal, the -round compile option can be used to change the value to hexadecimal. However, values specified in hexadecimal can be fixed without specifying this option.

```
When a floating-point number is specified in
decimal

float x = 0.1f;

Changing the value by option

-round=zero     : 0x3DCCCCCC
-round=nearest  : 0x3DCCCCCD
```

```
When a floating-point number is specified in
hexadecimal

float x = 0x0.CCCCCCP-3;

Changing the value by option

-round=zero     : 0x3DCCCCCC
-round=nearest  : 0x3DCCCCCC
```

2.3.2 enum

■ Overview

Compilation is now performed correctly, even when a trailing comma is included in an enumeration type.

■ Description

Under the C89 specification, an error occurs when an extra comma exists at the end of an enum declaration. However, the specification in C99 has been changed so that no error occurs.

Note that the RX-family C/C++ compiler allows extra commas, even under C89.

■ Example usage

No compile error occurs when a comma exists after CCC, the last item in an enum declaration.

```
enum E { AAA, BBB, CCC, };

enum E e = CCC;
```

2.3.3 Array and structure initialization

■ Overview

Arrays and structures can now be initialized with specific elements or members.

■ Description

In C99, the element number and structure members of an array can be specified explicitly by using a format called a *specification initializer*. When an array is initialized in C89, each element needs to be initialized in order from the beginning. In C99, initial values can be set only for certain elements, with uninitialized elements being initialized as 0. Like structure members, members need to be initialized for C89 in order from the beginning. In C99, initial values can be set only for specific members, with uninitialized members being initialized as 0. This is useful when the elements or members requiring initialization are limited, or when arrays have large element counts or structures with many members.

■ Example usage

(1) Example usage for arrays

In this example, the initial values are specified for the third and fourth elements in the array. Other values are set to 0.

C89

```
int array[5] = { 0, 0, 0, 2, 1 };
```

C99

```
int array[5] = { [3] = 2, [4] = 1 };
```

(2) Example usage for structures

In this example, initial values are set only for specific structure members. This is also possible for nested structures such as structure T. Initialization can be limited to specified members, such as for structures like S1, which has many nested members.

C89

```
struct S {
    int a;
    int b;
} s = { 0, 1 };

struct T {
    int a;
    int b;
    struct T1 {
        int aa;
        int bb;
    } t1;
} t = { 0, 1, { 0, 2 } };

struct S1 {
    int a;
    int b[100];
    int c;
} s1 = { 10, {0,0,...}, 20 };
```

C99

```
struct S {
    int a;
    int b;
} s = { .b = 1 };

struct T {
    int a;
    int b;
    struct T1 {
        int aa;
        int bb;
    } t1;
} t = { .b = 1, .t1.bb = 2 };

struct S1 {
    int a;
    int b[100];
    int c;
} s1 = { .a = 10, .c = 20 };
```

2.3.4 Compound literals

■ Overview

Structure and array types can be specified as immediate values.

■ Description

C99 allows the creation of anonymous objects with initial values. This allows more concise code for handling initialized array data and structure data. The compound literal format is as follows:

(type [element count]){ element1, element2, ... }

■ Example usage

In C89, the temp Point type variable needs to be declared, have its members initialized, and then be passed to the function func. However, in C99, values can be passed directly to functions without creating temporary variables like temp.

C89

```
typedef struct Point {
    short x,y;
} Point;

Point x;

void func(Point *p)
{
    x = *p;
}

void func_1()
{
    Point temp = {100,200};
    func(&temp);
}

void func_2()
{
    Point temp[2] = {{100,200},{300,400}};
    func(temp);
}
```

C99

```
typedef struct Point {
    short x,y;
} Point;

Point x;

void func(Point *p)
{
    x = *p;
}

void func_1()
{

    func(&(Point){100,200});
}

void func_2()
{

    func((Point[2]){{100,200}, {300,400}});
}
```

2.4 Syntax

The following explains the syntactical additions to C99.

2.4.1 One-line comments

■ Overview

One-line comments can now be used in C++.

■ Description

Until C89, comments could only be used with the /**/ format. In C99, lines that start with // and end with a line return can be used as one-line C++ comments.

Note that the RX-family C/C++ compiler supports // comments in C89 as well.

■ Example usage

The following example uses a one-line comment.

C89

```
void func()
{
    int a;
    a = 5;    /* Comment */
}
```

C99

```
void func()
{
    int a;
    a = 5;    // Comment
}
```

2.4.2 Wide character concatenation

■ Overview

The method for concatenating wide characters has been formalized.

■ Description

With the C language, when two or more strings are coded consecutively, they are concatenated into a single string. For example, when the strings "aaa" "bbb" (two strings separated by a half-width space) are specified, they are treated as the concatenated string "aaabbb". Likewise, when wide strings such as L"aaa" L"bbb" are coded consecutively, they are handled as the wide string L"aaabbb". In C89, the operation for concatenating consecutive strings and wide strings is not defined, but the specification for C99 dictates that when a string and wide string exist consecutively, they are concatenated as a wide string.

Note that the C6282 error is thrown by the RX-family C/C++ compiler for C89 when a string and wide string are specified consecutively.

■ Example usage

The string "Application" and wide string "Note" are concatenated and handled as L"ApplicationNote".

```
#include <wchar.h>

wchar_t *str = "Application" L"Note";
```

2.4.3 Variable arity macros

■ Overview

Macros can now be used with variable arity.

■ Description

In C89, variable arity could be used for functions such as printf and scanf, but not for macros. In C99, variable arity can also be used for macros. The format is as follows:

#define macro-name (str, ...) Specify __VA_ARGS__ in locations that use variable arity

Note that when using a macro, variable arity can also be omitted.

■ Example usage

In C89, macros with different argument counts such as DEBUG1 and DEBUG2 needed to be defined separately. In C99, macros with different argument counts can be completed with one definition.

```
C89

#include <stdio.h>

#define DEBUG1(fmt, val1)       printf("[debug] : " ## fmt, val1);
#define DEBUG2(fmt, val1, val2) printf("[debug] : " ## fmt, val1, val2);

void func()
{
   int a = 0;
   int b = 1;

   DEBUG1("a = %d¥n", a);
   DEBUG2("a = %d¥n, b = %d", a, b);
}
```

```
C99

#include <stdio.h>

#define  DEBUG(fmt, ...)  printf("[debug] : " ## fmt, __VA_ARGS__);

void func()
{
   int a = 0;
   int b = 1;

   // Interpreted as printf("[debug] : a = %d¥n", a);
   DEBUG("a = %d¥n", a);

   // Interpreted as printf("[debug] : a = %d, b = %d¥n", a, b);
   DEBUG("a = %d, b = %d¥n", a, b);

   // Specifications such as the following are also possible.
   // Interpreted as printf("[debug] : message¥n");
   DEBUG("message¥n");
}
```

2.4.4 Empty arguments in function type macros

■ Overview

Empty arguments can now be passed to function type macros.

■ Description

In C89, arguments could not be omitted when function type macros were used. In C99, empty arguments can now be passed.

Note that the RX-family C/C++ compiler does not output an error for C89, but a warning (C5054). As in C99, this is handled as an empty specification.

■ Example usage

The following example shows both passing all arguments to the MESSAGE function type macro and passing only some arguments.

```
#include <stdio.h>

#define MESSAGE(msg1, msg2)   "msg1 is " ## msg1 ## ". msg2 is " ## msg2 ## "."

void func()
{
   // Interpreted as printf("%s¥n", "msg1 is AAA. msg2 is BBB.");
   printf("%s¥n", MESSAGE("AAA", "BBB"));

   // Interpreted as printf("%s¥n", "msg1 is CCC. msg2 is .");
   printf("%s¥n", MESSAGE("CCC"));
}
```

2.4.5 Characters that can be used in identifiers

■ Overview

Universal characters and multi-byte characters can now be used in identifiers.

■ Description

In C89, the characters that could be used as identifiers were limited to alphanumerics (uppercase and lowercase characters), numerals (except for the beginning of the identifier), and underscores (_). In C99, the characters that can be used as identifiers have been extended to include universal characters and multi-byte characters. However, numerals still cannot be used at the beginning of an identifier.

■ Example usage

The following example uses a variable and function named using the Japanese words for "variable" and "function," respectively.

```
char ¥u6587¥u5B57;  /* A variable named "character" in Japanese */

void func()
{
   ¥u6587¥u5B57 = 'a';
}

// A function named "function" in Japanese
void ¥u95A2¥u6570()
{
   func();
}

void main()
{
   ¥u95A2¥u6570();
}
```

2.4.6 Variable declaration positions

■ Overview

Variables can now be declared in the middle of a block.

■ Description

In C89, variables had to be declared at the beginning of a block. From C99, variables can be declared after the beginning of the block, but only before they are referenced.

■ Example usage

The private variables `i` and `j` are declared in the function func, but in C99, the variables can be declared in any position within the function.

```
C89

int a[10], b[10];

void func()
{
    int i;
    int j;

    for (i=0; i<10; i++) {
       a[i] = 10;
    }


    for (j=0; j<10; j++) {
       b[j] = j;
    }
}
```

```
C99

int a[10], b[10];

void func()
{
    int i;


    for (i=0; i<10; i++) {
       a[i] = 10;
    }

    int j;
    for (j=0; j<10; j++) {
       b[j] = j;
    }
}
```

2.5 Standard include files

Six standard include files have been added in C99: complex.h, fenv.h, inttypes.h, stdbool.h, stdint.h, and tgmath.c.

2.5.1 complex.h

■ Overview

A complex number calculation library has been added.

■ Description

complex.h is a standard include file for using the complex number calculation library. The following table lists the functions that complex.h contains. For float type complex numbers, the definition name is the function name with "f" appended, for long double type complex numbers, the definition name is the function name with "l" appended, and for double type complex numbers, the definition name is the same as the function name.

The -head=complex option needs to be specified in the standard library configuration tool in order to use the complex number calculation library.

Table 2-4 Complex number calculation library

Type	Definition name	Description
Function	cacos	Calculates the arc cosine of a complex number.
	casin	Calculates the arc sine of a complex number.
	catan	Calculates the arc tangent of a complex number.
	ccos	Calculates the cosine of a complex number.
	csin	Calculates the sine of a complex number.
	ctan	Calculates the tangent of a complex number.
	cacosh	Calculates the arc hyperbolic cosine of a complex number.
	casinh	Calculates the arc hyperbolic sine of a complex number.
	catanh	Calculates the arc hyperbolic tangent of a complex number.
	ccosh	Calculates the hyperbolic cosine of a complex number.
	csinh	Calculates the hyperbolic sine of a complex number.
	ctanh	Calculates the hyperbolic tangent of a complex number.
	cexp	Calculates the natural logarithm of base e to the power of z for a complex number.
	clog	Calculates the natural logarithm of a complex number.
	cabs	Calculates the absolute value of a complex number.
	cpow	Calculates the exponent of a complex number.
	csqrt	Calculates the square root of a complex number.
	carg	Calculates a phase angle.
	cimag	Calculates an imaginary component.
	conj	Reverses the sign of the imaginary component and calculates the complex conjugate.
	cproj	Calculates a projection on a Riemann sphere.
	creal	Calculates a real number.

■ Example usage

The following example calculates the arc cosine of a complex number.

```
#include <complex.h>

double complex z, ret;

void func(void)
{
   ret = cacos(z);
}
```

2.5.2 fenv.h

■ Overview

A floating-point environment library has been added.

■ Description

fenv.h is a standard include file for accessing a floating-point environment. A floating-point environment includes the floating-point status flag, floating-point exception flag, and other flags defined in fenv.h. The following lists the macros and functions in fenv.h.

The -head=fenv option needs to be specified in the standard library configuration tool in order to use the floating-point environment library.

Table 2-5 Floating-point environment library

Type	Definition name	Description
Type (macro)	fenv_t	The type for the entire floating-point environment.
	fexcept_t	The type for a floating-point status flag.
Constant (macro)	FE_DIVBYZERO	Macros defined when floating-point exceptions are supported.
	FE_INEXACT	
	FE_INVALID	
	FE_OVERFLOW	
	FE_UNDERFLOW	
	FE_ALL_EXCEPT	
Constant (macro)	FE_DOWNWARD	Macros for the rounding direction of floating-point numbers.
	FE_TONEAREST	
	FE_TOWARDZERO	
	FE_UPWARD	
Constant (macro)	FE_DEF_ENV	An existing floating-point environment for programs.
Function	feclearexcept	Attempts to clear a floating-point exception.
	fegetexceptflag	Attempts to store a value in the status object of a floating-point flag.
	feraiseexcept	Attempts to generate a floating-point exception.
	fesetexceptflag	Attempts to set a floating-point flag.
	fetestexcept	Checks whether a floating-point flag is set.
	fegetround	Obtains the rounding direction.
	fesetround	Sets the rounding direction.
	fegetenv	Attempts to obtain a floating-point environment.
	feholdexcept	Saves the floating-point environment, clears the floating-point status flag, and sets the non-hold mode for floating-point exceptions.
	fesetenv	Attempts to set a floating-point environment.
	feupdateenv	Attempts to save a floating-point exception to the automatic memory area, set a floating-point environment, and generate the saved floating-point exception.

■ Example usage

The following example attempts to clear a floating-point exception.

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
int ret, e;

void func()
{
    ret = feclearexcept(e);
}
```

2.5.3 inttypes.h

■ Overview

A library for converting integer type formats has been added.

■ Description

inttypes.h is a standard include file for extending integer types. It also provides a format for handling integer types defined in stdint.h. The following table lists its macros and functions.

The -head=inttypes option must be specified for the standard library configuration tool in order to use the format conversion library.

Table 2-6 Format conversion library for integer types

Type	Definition name	Description
Type (macro)	imaxdiv_t	The type for values returned by the imaxdiv function.
Variable (macro)	PRId*N*	The format of the printf function.
	PRIdLEAST*N*	
	PRIdFAST*N*	
	PRIdMAX	
	PRIdPTR	
	PRIi*N*	
	PRIiLEAST*N*	
	PRIiFAST*N*	
	PRIiMAX	
	PRIiPTR	
	PRIo*N*	
	PRIoLEAST*N*	
	PRIoFAST*N*	
	PRIoMAX	
	PRIoPTR	
	PRIu*N*	
	PRIuLEAST*N*	
	PRIuFAST*N*	
	PRIuMAX	
	PRIuPTR	
	PRIx*N*	
	PRIxLEAST*N*	
	PRIxFAST*N*	
	PRIxMAX	
	PRIxPTR	
	PRIX*N*	
	PRIXLEAST*N*	
	PRIXFAST*N*	
	PRIXMAX	
	PRIXPTR	
	SCNd*N*	
	SCNdLEAST*N*	
	SCNdFAST*N*	
	SCNdMAX	
	SCNdPTR	
	SCNi*N*	
	SCNiLEAST*N*	
	SCNiFAST*N*	
	SCNiMAX	

	SCNiPTR	
	SCNo*N*	
	SCNoLEAST*N*	
	SCNoFAST*N*	
	SCNoMAX	
	SCNoPTR	
	SCNu*N*	
	SCNuLEAST*N*	
	SCNuFAST*N*	
	SCNuMAX	
	SCNuPTR	
	SCNx*N*	
	SCNxLEAST*N*	
	SCNxFAST*N*	
	SCNxMAX	
	SCNxPTR	
function	imaxabs	Calculates absolute values.
	imaxdiv	Calculates quotients and remainders.
	strtoimax	Aside from converting the start of a string to an intmax_t type or uintmax_t type expression, this is the same as the strtol, strtoll, strtoul, and strtoull functions.
	strtoumax	
	wcstoimax	Aside from converting the start of a wide string to an intmax_t type or uintmax_t type expression, this is the same as the wcstol, wcstoll, wcstoul, and wcstoull functions.
	wcstoumax	

Note: *N* is one of 8, 16, and 32, and may also be 64 due to macros.

■ Example usage

The following example outputs variable a with the int_latest32_t type. The result of printf is "value : 30".

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

void func()
{
   int_least32_t a;
   a = 30;
   printf("value : %"PRIdLEAST32"¥n", a);
}
```

2.5.4 stdbool.h

■ Overview

A standard include file for defining macros related to logical types and logical values has been added.

■ Description

The bool, true, and false names can be used. stdbool.h is an include file that consists only of macro name definitions. The following table lists these definitions.

Table 2-7 Logical type macros

Type	Definition name	Description
Macro (variable)	bool	Expands to _Bool.
Macro (constant)	true	Expands to 1.
	false	Expands to 0.
	__bool_true_false_are_defined	Expands to 1.

■ Example usage

The following example uses stdbool.h.

```
#include <stdbool.h>

bool a;
int func()
{
    if (a == true) {
        return 1;
    }
    return 0;
}
```

2.5.5 stdint.h

■ Overview

A standard include file for declaring integer types of a specified width has been added.

■ Description

Integer types are defined according to each condition. Since the size of an integer type does not depend on the environment, integer types are highly portable across environments. stdint.h is an include file that consists only of macro name definitions. The following table lists these definitions.

Table 2-8 Macros for specified-width integer type

Type	Definition name	Description
Macro	int_least*N*_t	A type that can store at least the size of each signed/unsigned integer type for 8, 16, 32, and 64 bits.
	unt_least*N*_t	
	int_fast*N*_t	A type that can calculate at maximum speed each signed/unsigned integer type for 8, 16, 32, and 64 bits.
	uint_fast*N*_t	
	intptr_t	A signed/unsigned integer type that can perform round-trip conversion for pointers to void.
	uintptr_t	
	intmax_t	A signed/unsigned integer type that can represent all values for all signed/unsigned integer types.
	uintmax_t	
	int*N*_t	A signed/unsigned integer type with a width of N bits.
	uint*N*_t	
	INT*N*_MIN	The minimum value of a width-specified signed integer type.
	INT*N*_MAX	The maximum value of a width-specified signed integer type.
	UINT*N*_MAX	The maximum value of a width-specified unsigned integer type.
	INT_LEAST*N*_MIN	The minimum value of the minimum-width-specified signed integer type.
	INT_LEAST*N*_MAX	The maximum value of the minimum-width-specified signed integer type.
	UINT_LEAST*N*_MAX	The maximum value of a minimum-width-specified unsigned integer type.
	INT_FAST*N*_MIN	The minimum value of the fastest minimum-width-specified signed integer type.
	INT_FAST*N*_MAX	The maximum value of the fastest minimum-width-specified signed integer type.
	UINT_FAST*N*_MAX	The maximum value of the fastest minimum-width-specified unsigned integer type.
	INTPTR_MIN	The minimum value of a signed integer type able to hold a pointer.
	INTPTR_MAX	The maximum value of a signed integer type able to hold a pointer.
	UINTPTR_MAX	The maximum value of an unsigned integer type able to hold a pointer.
	INTMAX_MIN	The minimum value of the maximum-width signed integer type.
	INTMAX_MAX	The maximum value of the maximum-width signed integer type.
	UINTMAX_MAX	The maximum value of the maximum-width unsigned integer type.
	PTRDIFF_MIN	-65535
	PTRDIFF_MAX	+65535
	SIG_ATOMIC_MIN	-127
	SIG_ATOMIC_MAX	+127
	SIZE_MAX	65535
	WCHAR_MIN	0
	WCHAR_MAX	65535U
	WINT_MIN	0
	WINT_MAX	4294967295U
Function (macro)	INT*N*_C	Expands Int_least*N*_t to an integer constant expression.
	UINT*N*_C	Expands uInt_least*N*_t to an integer constant expression.
	INT_MAX_C	Expands intmax_t to an integer constant expression.
	UINT_MAX_C	Expands uintmax_t to an integer constant expression.

Note: *N* is one of 8, 16, and 32, and may also be 64 due to macros.

■ Example usage

In the following example, variable `a` is the smallest possible type able to store a 32-bit signed integer. For the RX-family C/C++ compiler, this is the long type.

```
#include <stdint.h>

int_least32_t a;
```

2.5.6 tgmath.h

■ Overview

A standard include file for defining generally named macros has been added.

■ Description

When tgmath.h is included and the mathematical functions listed in the following table (generally named macros) are used, they are automatically expanded to function names according to the argument type. For example, when a float type argument is passed to the sin function, it is expanded to the sinf function, and when a complex type argument is passed, it is expanded to the csin function. tgmath.h is an include file that consists only of macro name definitions. The following table lists these definitions.

Table 2-9 Generally named macros

Generally named macro	math.h function	complex.h function
acos	acos	cacos
asin	asin	casin
atan	atan	catan
acosh	acosh	cacosh
asinh	asinh	casinh
atanh	atanh	catanh
cos	cos	ccos
sin	sin	csin
tan	tan	ctan
cosh	cosh	ccosh
sinh	sinh	csinh
tanh	tanh	ctanh
exp	exp	cexp
log	log	clog
pow	pow	cpow
sqrt	sqrt	csqrt
fabs	fabs	cfabs
atan2	atan2	-
cbrt	cbrt	-
ceil	ceil	-
copysign	copysign	-
erf	erf	-
erfc	erfc	-
exp2	exp2	-
expm1	expm1	-
fdim	fdim	-
floor	floor	-
fma	fma	-
fmax	fmax	-
fmin	fmin	-
fmod	fmod	-
frexp	frexp	-
hypot	hypot	-
ilogb	ilogb	-
ldexp	ldexp	-
lgamma	lgamma	-
llrint	llrint	-
llround	llround	-

log10	log10	-
log1p	log1p	-
log2	log2	-
logb	logb	-
lrint	lrint	-
lround	lround	-
nearbyint	nearbyint	-
nextafter	nextafter	-
nexttoward	nexttoward	-
remainder	remainder	-
remquo	remquo	-
rint	rint	-
round	round	-
scalbn	scalbn	
scalbln	scalbln	-
tgamma	tgamma	-
trunc	trunc	-
carg	-	carg
cimag	-	cimag
conj	-	conj
cproj	-	cproj
creal	-	creal

■ Example usage

In the following example, sin is automatically expanded to sinf.

```
#include <tgmath.h>

float f,ret;
void func(){
    ret = sin(f);
}
```

2.6 Macros

The following explains the macros that have been added to C99.

2.6.1 Predefined macros

■ Overview

The three macros __STDC_ISO_10646__, __STDC_IEC_559__, and __STDC_IEC_559_COMPLEX__ have been added.

■ Description

__STDC_ISO_10646__ is defined when character codes represented by wchar_t comply with ISO/IEC 10646. The time of ratification of the ISO/IEC 10646 specification to which the macro values conform is specified in yyyymmL format (where yyyy indicates the year and mm indicates the month). For the RX-family C/C++ compiler, 199712L is defined for C99.

__STDC_IEC_559__ indicates conformance to annex F (IEC60559 floating points). For the RX-family C/C++ compiler, this is defined for C99, but not for C89.

__STDC_IEC_559_COMPLEX__ indicates conformance to annex G (IEC60559-compatible complex numbers). For the RX-family C/C++ compiler, this is defined for C99, but not for C89.

■ Example usage

(1) __STDC_ISO_10646__

For the RX-family C/C++ compiler, 199712 is stored in variable `x` for C99, and 0 is stored for C89.

```
#ifdef  __STDC_ISO_10646__
unsigned long x = __STDC_ISO_10646__;
#else
unsigned long x = 0;
#endif
```

(2) __STDC_IEC_559__

For the RX-family C/C++ compiler, compilation can be performed for C99, but a compile error occurs for C89.

```
#ifdef __STDC_IEC_559__

#else
#error incompatible float
#endif
```

(3) __STDC_IEC_559_COMPLEX__

For the RX-family C/C++ compiler, compilation can be performed for C99, but a compile error occurs for C89.

```
#ifdef __STDC_IEC_559_COMPLEX__

#else
#error incompatible complex
#endif
```

2.6.2 __func__

■ Overview

This is a macro that is replaced by the name of the function in the specified position.

■ Description

__func__ is replaced by the name of the function in the specified position. It can also be used as a macro, in which case it is replaced with the name of the function using the macro.

■ Example usage

__func__ is replaced with "func_001" for function func_001, and with "func_002" for function func_002.

```
#include<stdio.h>

void func_001()
{
    /* "function : func_001" is output. */
    printf("function : %s¥n",__func__);
}

void func_002()
{
    /* "function : func_002" is output. */
    printf("function : %s¥n",__func__);
}
```

The following example uses __func__ for a macro. The results are the same as for the above example.

```
#include<stdio.h>

#define DEBUG()  printf("function : %s¥n", __func__)

void func_001()
{
    /* "function : func_001" is output. */
    DEBUG();
}

void func_002()
{
    /* "function : func_002" is output. */
    DEBUG();
}
```

2.7 Pragmas

The following explains the pragmas added to C99.

2.7.1 #pragma STDC FP_CONTRACT

■ Overview

#pragma STDC FP_CONTRACT can be used to control whether floating-point arithmetic is omitted.

■ Description

When floating-point arithmetic is omitted, the margin of error from rounding floating-point constant values is not taken into consideration, and any exceptions that occur due to loss of precision as a result of arithmetic are not reported. When #pragma STDC FP_CONTRACT ON is declared, subsequent floating-point arithmetic is permitted to be omitted. Likewise, #pragma STDC FP_CONTRACT OFF can be declared to prohibit omission of subsequent floating-point arithmetic. When #pragma STDC FP_CONTRACT DEFAULT is declared, omission of subsequent floating-point arithmetic returns to the default specification. Whether the default specification is ON or OFF depends on the processing type definition.

For the RX-family C/C++ compiler, any #pragma STDC FP_CONTRACT specifications are disregarded.

■ Example usage

The following example uses #pragma STDC FP_CONTRACT to control whether floating-point arithmetic omission is permitted.

```
float x,y,z;

#pragma STDC FP_CONTRACT ON
/* Floating-point arithmetic omission is permitted. */

void func1()
{
   x = y / z;
}

#pragma STDC FP_CONTRACT OFF
/* Floating-point arithmetic omission is prohibited. */

void func2()
{
   x = y / z;
}

#pragma STDC FP_CONTRACT DEFAULT
/* Floating-point arithmetic omission follows the default. */

void func3()
{
   x = y / z;
}
```

2.7.2 #pragma STDC FENV_ACCESS

■ Overview

#pragma STDC FENV_ACCESS can be used to control access to floating-point environments.

■ Description

A floating-point environment is one with the floating-point status flag, floating-point exception flag, and other flags defined in the filefenv.h standard include. By explicitly notifying the compiler whether access to floating-point environments is performed, the compiler can more easily perform optimizations for floating-point environments. When #pragma STDC FENV_ACCESS ON is declared, the compiler is notified that access to floating-point environments may occur subsequently. Conversely, when #pragma STDC FENV_ACCESS OFF is declared, the compiler is notified that access to floating-point environments will not occur subsequently. When #pragma STDC FENV_ACCESS DEFAULT is declared, whether access to floating-point environments occurs subsequently returns to the default specification. Whether the default specification is ON or OFF depends on the processing type definition.

For the RX-family C/C++ compiler, access to floating-point environments is always assumed to be possible, so any #pragma STDC FENV_ACCESS specifications are disregarded.

■ Example usage

The following example uses #pragma STDC FENV_ACCESS to control whether access to floating-point environments exists.

```
#include <fenv.h>

long a, b;

#pragma STDC FENV_ACCESS ON
/* Access to floating-point environments is performed. */

void func1()
{
   a = feraiseexcept(b);
}

#pragma STDC FENV_ACCESS OFF
/* Access to floating-point environments is not performed. */

void func2()
{
   a = feraiseexcept(b);
}

#pragma STDC FENV_ACCESS DEFAULT
/* Access to floating-point environments follows the default. */

void func3()
{
   a = feraiseexcept(b);
}
```

2.7.3 #pragma STDC CX_LIMITED_RANGE

■ Overview

#pragma STDC CX_LIMITED_RANGE can be used to control mathematical formulas for complex number arithmetic.

■ Description

Multiplication and division of complex numbers, as well as mathematical formulas for absolute values, present problems due to the handling of infinite values and improper overflows and underflows. #pragma STDC CX_LIMITED_RANGE can be used to notify the compiler that mathematical formulas may be applied. When #pragma STDC CX_LIMITED_RANGE ON is declared, mathematical formulas may be applied subsequently. Conversely, when #pragma STDC CX_LIMITED_RANGE OFF is declared, mathematical formulas may not be applied subsequently. When #pragma STDC CX_LIMITED_RANGE DEFAULT is declared, whether mathematical formulas may be applied subsequently follows the default specification. The default specification is OFF.

For the RX-family C/C++ compiler, mathematical formulas are always used, even when #pragma STDC CX_LIMITED_RANGE is specified.

■ Example usage

The following example uses #pragma STDC CX_LIMITED_RANGE to control whether mathematical formulas may be applied.

```
#include <complex.h>

float complex cf1, cf2, cf3;

#pragma STDC CX_LIMITED_RANGE ON
/* Mathematical formulas may be applied. */

void func1()
{
   cf1 = cf2 * cf3;
}

#pragma STDC CX_LIMITED_RANGE OFF
/* Mathematical formulas may not be applied. */

void func2()
{
   cf1 = cf2 * cf3;
}

#pragma STDC CX_LIMITED_RANGE DEFAULT
/* Whether mathematical formulas may be applied follows the default. */

void func3()
{
   cf1 = cf2 * cf3;
}
```

3. Precautions regarding migration from C89 to C99

3.1 Implicit type declarations

■ Overview

The specification regarding implicit function declarations and implicit type declarations has changed.

■ Description

In C89, functions could be called without function prototype declarations. In the C99 specification, operation when these functions are called is now undefined.

Also, in C89, implicit type declarations were handled as int types. In the C99 specification, implicit variable declarations are no longer permitted.

■ Example usage

(1) Implicit function calls

In the following example, the function func calls the function printf, for which no prototype has been declared (originally, stdio.h needed to be included). When the RX-family C/C++ compiler compiles for C89 and the -message option is specified, C5223 is output at the information level, but when it compiles for C99, C5223 is output at the warning level.

```
void func(void)
{
    printf("message");
}
```

(2) Implicit type declarations

In the following example, the types for the return value of the function func and the temporary argument `a` are each treated as int types in C89. When the RX-family C/C++ compiler compiles for C89 and the -message option is specified, C5260 (an informational message) is output because the type of return value is not declared. When it compiles for C99, C5260 (an informational message) is output because the type of return value is not declared, and C6051 is output at the warning level because no type is declared for the temporary argument `a`. In the following example, the function is interpreted as int func(int a) in both C89 and C99.

```
func(a)
{
    return a;
}
```

3.2 Negative integer division

■ Overview

The behavior for negative integer division may sometimes differ.

■ Description

In C89, if either the divisor or dividend is negative for integer division, the results differed depending on the implementation. In C99, however, the results for all types of division are rounded down. Accordingly, the behavior of programs containing negative integer division may differ.

For the RX-family C/C++ compiler, results are also rounded down for C89.

■ Example usage

A calculated result of -1/2 is -0.5, but for C89, the compiler may calculate it as -1 or 0. For C99 the result is always calculated as 0.

```
int x = -1/2;
```

Website and Support

- Renesas Electronics Website
  http://www.renesas.com/
- Inquiries
  http://www.renesas.com/inquiry

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Apr.20.10	—	First edition issued
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.
   - "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.
   - "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.
   - "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

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

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

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
7F, No. 363 Fu Shing North Road Taipei, Taiwan, R.O.C.
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

© 2010 Renesas Electronics Corporation. All rights reserved.

Colophon 1.0