SPIRIT Platform Blueprint

SPIRIT C Language Portability Guide

(SPIRIT Issue 3.0)

Network Management Forum

SPIRIT Platform Blueprint

SPIRIT C Language Portability Guide (SPIRIT Issue 3.0)

ISBN: N/A
Document Number: J407

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to:

Network Management Forum
1201 Mount Kemble Avenue
Morristown, NJ 07960
U.S.A.

Tel: +1 201 425 1900

Contents

Chapter 1

Introduction

1.1 Purpose

This document is intended to make application programs based on the SPIRIT C-language interface specifications (SPIRIT C) more portable.

SPIRIT C is defined to improve the portability of application programs by eliminating the differences among implementations as far as possible. Therefore, a SPIRIT C application program may be ported with little modification. The application program will necessarily contain some contrived coding in order to make the implementation-defined portion more portable.

The purpose of this document is to improve the portability of the implementation-defined portion.

Implementations may include extensions beyond the range defined by SPIRIT — these are not described here.

1.2 Programming Technique

This section describes the kinds of technique available for a program for which the wording ‘‘it should be rewritten’’ is specified in the guide:

1. Cases where changes are made on the word basis.

   Replace words with the statement below, and gather occurrences of the statements into a specific area in the source program:

   C: #define preprocessing directive

2. Cases where changes are made on the line basis.

   Separate the lines to be changed and then include them into the source program.

   C: #include preprocessing directive

3. Cases where changes are made on the execution basis.

   Make subroutines from the parts to be changed.

   C: function

4. Cases of exception handling.

   Exception handling should be localised as described below, regardless of dependency on implementations.

   C: use a *signal*( ) function to specify the place where the post-process is written

5. Cases other than the above.

   When the methods shown above are not applicable, comments for rewriting information should be written *in situ*. How to write a comment is described below.

C: From "/*" to "*/"

Limit Values

In the interface specifications, limit values are defined for each interface. An application program which exceeds those limit values may be executed properly, but is not portable. Application programs should be written within the limit value.

1.3 How to Read this Document

Each section of the guide is composed of the following items:

NAME

<type of guide>-<classification number>-<sequence number>

<type of guide> is C (C Language).
<classification number> is a number assigned to CLASSIFICATION.
<sequence number> is a number within the CLASSIFICATION.

CLASSIFICATION

Classification based on the content.

TITLE

A title which represents the content.

CLAUSE

The section/chapter number and its title of the corresponding specification.

GUIDANCE

The matters to be followed in order to improve portability.

EXPLANATION

Reason why it should be done in this way.

EXAMPLE

An example which shows an application program without portability, if necessary, and a way to improve portability.

Category for Which a Solution is Available

This category includes those programs for which, although implementation-defined, there is a solution, such as a coding method; for example, order of evaluation. The guidance for a case like this is expressed as ‘‘shall’’ or ‘‘should’’ for a requirement, and ‘‘shall not’’ or ‘‘should not’’ for a prohibition.

In EXAMPLE, the example of an application program of low portability is shown in Application Program Without Portability, and the portion which varies depending on implementation is explained. In Application Program With Portability, as a solution which does not depend on the implementation, an example of an application program of high portability is shown.

Category for Which Rewriting is Required

An application program in which names, such as file names, or processing method, such as input/output, vary with implementations and which requires rewriting belong to this category.

The guidance for this category is expressed as ‘‘shall be rewritten’’ or ‘‘should be rewritten’’.

In EXAMPLE, the example of an application program of low portability is shown in Application Program Without Portability, and the portion which is not portable is explained.

Others

Reserved words, for example, are listed for reference in EXAMPLE.

Chapter 2

Application Program Portability Guide

NAME

C-1-1

CLASSIFICATION

Statements and Preprocessing Directives

TITLE

Use of the same label

CLAUSE

6.1 Lexical Elements

GUIDANCE

The same name should not be used as a label more than once within the same function.

EXPLANATION

When a label which has the same name is declared more than once within the same function, the behaviour is not defined. Therefore, the behaviour for this situation varies with the implementation. Some treat it as a compilation error, others ignore the second and subsequent labels. The identifiers of the same name may be used as labels within the same source program, provided that they appear within different functions. This means that the label is referenced only within the function where it is defined.

EXAMPLE

Application Program Without Portability

In the example below, if A and B are #defined, the label error: will appear twice.

```
#ifdef A
    /* process of A */
    error:
    ...
#endif /* A */
#ifdef B
    /* process of B */
    error:
    ...
#endif /* B */
```

NAME

C-1-2

CLASSIFICATION

Statements and Preprocessing Directives
Character String Handling

TITLE

Character constants during preprocessing and execution

CLAUSE

6.1.3.4	Character Constants
6.8.1	Conditional Inclusion

GUIDANCE

Characters should not be compared by preprocessing directives, and care should be used when the character is accessed by a value.

EXPLANATION

In SPIRIT, the value of characters is implementation-defined, except that SPIRIT defines that when 0123456789 identifies the numbers, each number must have a value greater by one than the previous number. Therefore, an application program which assumes the relation of characters of a different class is not portable.

EXAMPLE

Application Program Without Portability

In the example below, when the code used by the implementation is ASCII code, it is true, because ’A’ == 0x41, ’0’ == 0x31. On the other hand, when the code used is EBCDIC code, it is false, because ’A’ == 0xC1, ’0’ == 0xF0. So, it is not portable.

```
#if ’A’ > ’0’
#endif
```

NAME

C-1-3

CLASSIFICATION

Statements and Preprocessing Directives

TITLE

Generation of ‘‘defined’’ wording

CLAUSE

6.8.1 Conditional Inclusion

GUIDANCE

Be careful not to generate, intentionally or unintentionally, the wording of ‘‘defined’’ during the expansion of **#if** or **#elif** preprocessing directives.

EXPLANATION

Prior to evaluation of constant expressions, the **#if** or **#elif** preprocessing directive expands the preprocessing tokens. However, the behaviour is not defined when the token of ‘‘defined’’ is generated as a result of the expansion and when the format of the unary operator ‘‘defined’’ is not correct format prior to the macro replacement.

EXAMPLE

Application Program Without Portability

In the example below, though ‘‘defined(bar)’’ is expected, the result can be ‘‘defined(---)’’.

```
#define cat(a,b) a ## b
#define bar ---
#if cat(defined,(bar))
    /* do something */
#endif
```

NAME

C-1-4

CLASSIFICATION

Statements and Preprocessing Directives

TITLE

Comment following **#else** and **#endif**

CLAUSE

6.8.1 Conditional Inclusion

GUIDANCE

Comment must not be used following **#else** and **#endif**.

EXPLANATION

Text used as a comment must not be used following **#else** and **#endif**, unless it is surrounded by comment brackets, /* */.

EXAMPLE

In the examples given below, the first one is incorrect, and the second one is permitted.

```
/* Incorrect */
#Ifdef TOKEN
    ...
#endif(Ifdef TOKEN)

/* Permitted */
#Ifdef TOKEN
    ...
#endif /*(ifdef TOKEN)*/
```

NAME

C-1-5

CLASSIFICATION

Statements and Preprocessing Directives

TITLE

The format of header name of **#include** is in error

CLAUSE

6.8.2 Source File Inclusion

GUIDANCE

Write the header names in standard format correctly.

EXPLANATION

The preprocessing token of the **#include** preprocessing directive may be macro expanded. However, the behaviour for the case where the result of the expansion is neither <filename> nor ‘‘filename’’ is not defined.

EXAMPLE

In the following example, INCLUDE_FILE will not be expanded to a form of <filename> or ‘‘filename’’. Therefore, it must be checked-out by the compiler or must follow the implementation-defined behaviour (for example, the preprocessing directive is ignored).

```
#define INCLUDE_FILE version6.h
#include INCLUDE_FILE
```

NAME

C-1-6

CLASSIFICATION

Statements and Preprocessing Directives

TITLE

Macro actual argument without preprocessing wording

CLAUSE

6.8.3 Macro Replacement

GUIDANCE

The actual argument of the function-like macro should not be null.

EXPLANATION

In SPIRIT, if the actual argument is null in a function-like macro invocation, the behaviour is undefined. Note that there are implementations which do not treat the null actual-argument as an error explicitly.

EXAMPLE

Application Program Without Portability

In the example below, the function-like macro *GetReq*() is defined having two dummy arguments, but those arguments are invoked in null.

```
#define SIZEOF(x) sz_##x
#define GetReq(name, req) if ((dpy->bufptr + SIZEOF(x##name##Req)) >
        dpy->bufmax) _XFlush(dpy); dpy->bufptr += SIZEOF(x##name##Req);
        dpy->request++
    ...
GetReq();
```

Some implementations determine this as an error, and others continue processing by expanding this as shown below on the assumption that both arguments are null.

```
if ((dpy->bufptr + sz_xReq) > dpy->bufmax)
    _XFlush(dpy);
dpy->bufptr += sz_xReq;
dpy->request++
```

NAME

C-1-7

CLASSIFICATION

Statements and Preprocessing Directives

TITLE

Macro actual argument with preprocessing directive line format

CLAUSE

6.8.3 Macro Replacement

GUIDANCE

This type of macro actual argument should not be provided.

EXPLANATION

If an actual argument is provided in the wording of the preprocessing directive line format in a function-like macro invocation, the behaviour is undefined. Note that some implementations handle this case on the assumption that the argument of null character string has been passed.

EXAMPLE

In the following example, fixed behaviour may not be guaranteed, because the actual argument of the function-like macro F includes a preprocessing directive of "#include".

```
#define F(X) {X}
    ...
F(
#include <stddef.h>
)
```

NAME

C-1-8

CLASSIFICATION

Statements and Preprocessing Directives

TITLE

Order of processing of the preprocessing operators, # and ##

CLAUSE

6.8.3.2 The # Operator

GUIDANCE

The preprocessing operators, # and ##, should not be used together.

EXPLANATION

The evaluation order of # and ## is unspecified. Therefore, these preprocessing operators should not be used in a way that may produce a different result depending upon the evaluation order of the operators.

EXAMPLE

Application Program Without Portability

With the following function-like macro, the reference of g(1) results in "1"y if # is processed first, and it results in the value of #1y; that is, "1y", if ## is processed first.

```
#define g(x) #x ##y
```

NAME

C-1-9

CLASSIFICATION

Statements and Preprocessing Directives

TITLE

Class and behaviour of a **#pragma** preprocessing directive

CLAUSE

6.8.6 Pragma Directive

GUIDANCE

An application program which uses a **#pragma** preprocessing directive must be rewritten when it is transferred to other implementations.

EXPLANATION

The class and behaviour of **#pragma** is implementation-defined. Some implementations do not support **#pragma**.

EXAMPLE

Application Program Without Portability

The following **#pragma** directives are some of the examples of preprocessing directives which impair portability.

```
#pragma assembler
#pragma chars unsigned
#pragma map LongExternFunction,lextfun
```

The **#pragma** directives which are not related with generation of codes, such as a listing control directive, do not impair the portability of an application program in the implementation which does not support **#pragma**, because the **#pragma** is ignored in such an implementation.

NAME

C-1-10

CLASSIFICATION

Statements and Preprocessing Directives

TITLE

Value of NULL pointer

CLAUSE

7.1.6 Common Definitions **<stddef.h>**

GUIDANCE

To identify a null pointer, use NULL instead of a value of zero.

EXPLANATION

The type of the NULL pointer is implementation-defined. Therefore, describe the null pointer as "NULL", for ease of reading of the source code and in view of portability.

EXAMPLE

Application Program Without Portability

In the program given below, the error checking of *fopen*( ) results in a comparison of inconsistent types.

```
#include <stdio.h>
FILE *fp;
char *filename;
extern void notify(char *msg);
    ...
if ((fp = fopen(filename, "r")) == 0) {
    notify ("can’t open data file");
    exit (EXIT_FAILURE);
}
```

Application Program With Portability

Rewrite the *if* statement as follows. NULL is defined in **<stdio.h>**.

```
#include <stdio.h>
FILE *fp;
char *filename;
extern void notify(char *msg);
    ...
if ((fp = fopen(filename, "r")) == NULL) {
    notify("can’t open data file");
    exit (EXIT_FAILURE);
}
```

NAME

C-1-11

CLASSIFICATION

Statements and Preprocessing Directives

TITLE

Dummy arguments of *offsetof* macro

CLAUSE

7.1.6 Common Definitions **<stddef.h>**

GUIDANCE

A member of a bit-field should not be specified for dummy argument.

EXPLANATION

In SPIRIT, the behaviour is undefined when the structure member which is not addressable by the **&** operator (that is, a bit-field), is specified to the second argument of the macro of offsetof.

EXAMPLE

In the example given below, a fixed result is not guaranteed, because the second argument **b** of the offsetof refers to a bit-field.

```
struct foo {
    unsigned int a:3;
    unsigned int b:4;
    unsigned int c:4;
    unsigned int d:5;
    unsigned int e:10;
    unsigned int f:6;
};
    ...
size = offsetof(struct foo, b);
```

NAME

C-2-1

CLASSIFICATION

Identifiers, File Names and Header Names

TITLE

Identifiers (distinction of lower-case and upper-case letters)

CLAUSE

6.1.2 Identifiers

GUIDANCE

The names for external names (identifiers with external linkage) should not be used if they become the same name when the distinction between lower-case and upper-case is eliminated.

EXPLANATION

Although it is defined that the lower-case and upper-case letters are handled differently for internal names; for external names, it is implementation-defined and it is allowed not to distinguish the lower-case and upper-case letters. Therefore, external names should be determined on the assumption that there is no distinction between lower-case and upper-case. Otherwise, the application program may be non-transferrable.

EXAMPLE

Application Program Without Portability

```
extern int func(char);
extern int FUNC(char);
```

Some systems treat **func** and **FUNC** as the same name, and others treat them as different names. To make the application program portable, avoid using the same external names with the only difference being lower-case and upper-case.

NAME

C-2-2

CLASSIFICATION

Identifiers, File Names and Header Names

TITLE

Use of the reserved identifiers

CLAUSE

7.1.3 Reserved Identifiers

GUIDANCE

The identifier with the same name as an identifier reserved in that context (other than as allowed by Clause 7.1.3) should not be declared or defined, because the behaviour is undefined for those cases.

EXPLANATION

The reserved identifiers for each context are as follows:

1. All identifiers that begin with two underscores, and all identifiers that begin with an underscore and an upper-case letter in all contexts.
2. All external identifiers that begin with an underscore in the context used as identifiers with file scope in both the ordinary identifier and tag name spaces.
3. Each macro name listed in Clauses 7.1.4 through 7.16 and used in all contexts, if any of its associated headers is included.
4. All identifiers with external linkage listed in Clauses 7.1.4 through 7.16 used in all contexts as identifiers with external linkage.
5. Each identifier used in the context as an identifier with file scope in the same name space if any of its associated headers is included.

The following table contains the reserved identifiers which belong to 3. through 5. above.

Reserved Macro Names (3.)

Associated Header	Corresponding Identifiers
<errno.h>	EDOM, ERANGE, EILSEQ, *errno* (this can be defined as an external identifier), a macro name which starts with a letter E and the next letter is either a numeric or an upper-case letter.
<stddef.h>	NULL, offsetof
<assert.h>	NDEBUG
<locale.h>	LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME, NULL, a macro name which starts with letter LC and the next letter is an upper-case letter.
<math.h>	HUGE_VAL
<signal.h>	SIG_DFL, SIG_ERR, SIG_IGN, SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM, a macro name which starts with letter SIG or SIG_ and the next letter is an upper-case letter.
<stdarg.h>	*va_start*, *va_arg*
<stdio.h>	_IOFBF, _IOLBF, _IONBF, BUFSIZ, EOF, FILE, FILENAME_MAX, L_tmpnam, NULL, SEEK_CUR, SEEK_END, SEEK_SET, TMP_MAX, stderr, stdin, stdout
<stdlib.h>	EXIT_FAILURE, EXIT_SUCCESS, MB_CUR_MAX, NULL, RAND_MAX
<string.h>	NULL
<time.h>	CLOCK_PER_SEC, NULL

Reserved External Identifiers (4.)

Associated Header	Corresponding Identifiers
<errno.h>	*errno* (this can be defined in a macro), an identifier which starts with is or to and the next letter is a lower-case letter.
<ctype.h>	*isalnam*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *tolower*( ), *toupper*( )
<locale.h>	setlocale, localeconv
<math.h>	acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, frexp, idexp, log, log10, modf, pow, sqrt, ceil, fabs, floor, fmod, an identifier consists of one of the function names listed above and suffixed with an f or l.
<setjmp.h>	*setjmp*( ), *longjmp*( )
<signal.h>	*signal*( ), *raise*( )
<stdarg.h>	va_end
<stdio.h>	*remove*( ), *rename*( ), *tmpfile*( ), *tmpnam*( ), *fclose*( ), *fflush*( ), *fopen*( ), *freopen*( ), *setbuf*( ), *fprintf*( ), *fscanf*( ), *printf*( ), *scanf*( ), *sprintf*( ), *sscanf*( ), *vfprintf*( ), *vprintf*( ), *vsprintf*( ), *fgetc*( ), *fgets*( ), *fputc*( ), *fputs*( ), *getc*( ), *getchar*( ), *gets*( ), *putc*( ), *putchar*( ), *puts*( ), *ungetc*( ), *fread*( ), *fwrite*( ), *fgetpos*( ), *fseek*( ), *fsetpos*( ), *ftell*( ), *rewind*( ), *clearerr*( ), *feof*( ),

Associated Header	Corresponding Identifiers
	ferror( ), *perror*( )
<stdlib.h>	*atof*( ), *atoi*( ), *atol*( ), *strtod*( ), *strtol*( ), *strtoul*( ), *rand*( ), *srand*( ), *calloc*( ), *free*( ), *malloc*( ), *realloc*( ), *abort*( ), *atexit*( ), *exit*( ), *getenv*( ), *system*( ), *bsearch*( ), *qsort*( ), *abs*( ), *div*( ), *labs*( ), *ldiv*( ), *mblen*( ), *mbtowc*( ), *wctomb*( ), *mbstowcs*( ), *wcstombs*( ), an identifier which starts with str and the next letter is a lower-case letter.
<string.h>	*memcpy*( ), *memmove*( ), *strcpy*( ), *strncpy*( ), *strcat*( ), *strncat*( ), *memcmp*( ), *strcmp*( ), *strcoll*( ), *strncmp*( ), *strxfrm*( ), *memchr*( ), *strchr*( ), *strcspn*( ), *strpbrk*( ), *strrchr*( ), *strspn*( ), *strstr*( ), *strtok*( ), *memset*( ), *strerror*( ), *strlen*( ), an identifier which starts with either str, mem or wcs and the next letter is a lower-case letter.
<time.h>	*clock*( ), *difftime*( ), *mktime*( ), *time*( ), *asctime*( ), *ctime*( ), *gmtime*( ), *localtime*( ), *strftime*( )
<iso646.h>	and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq
<wchar.h>	**mbstate_t**, NULL, **size_t**, **struct tm**, **wchar_t**, WCHAR_MAX,WCHA_MIN, **wint_t**
<wctype.h>	wctrans_t, wctype_t, WEOF, wint_t(5)

Identifiers with File Scope (5.)

(The parenthesised phrase represents the corresponding name spaces.)

Associated Header	Corresponding Identifiers
<ctype.h>	**ptrdiff_t**, **size_t**, **wchar_t** (type names)
<locale.h>	lconv (tag name)
<setjmp.h>	**jmp_buf** (type name)
<signal.h>	**sig_atomic_t** (type name)
<stdarg.h>	*va_list* (type name)
<stdio.h>	**fpos_t**, **size_t** (type names)
<stdlib.h>	**div_t**, **ldiv_t**, **size_t**, **wchar_t** (type names)
<string.h>	**size_t** (type name)
<time.h>	**clock_t**, **time_t**, **size_t** (type names) tm (tag name)

NAME

C-2-3

CLASSIFICATION

Identifiers, File Names and Header Names

TITLE

Structure of file names for preprocessing directives, library function

CLAUSE

6.8.2	Source File Inclusion
7.9.3	Files

GUIDANCE

Rewriting of the file name for the **#include** preprocessing directive, library function may be required in some cases. It is desirable to use macros and modify only the macros on conversion, or if the target operating system is known beforehand, separate by macros and localise the portion to be modified at conversion time.

EXPLANATION

The association operation of the file name in the preprocessing directive, library function within a program to the actual file name is implementation-defined. Depending on the system, it may be permitted to add user ID to the file name in the source program, or to assign the member name to the file name.

EXAMPLE

Application Program Without Portability

```
#include "infile"
```

The file name recognised by the system is not necessarily ‘‘infile’’. Sometimes it is ‘‘uid.infile’’, or ‘‘uid.lib(infile)’’ with a member name. It must be rewritten according to the naming rule for file names of the target system.

Application Program With Portability

```
#if SYSTEM == ’a’
    #include "infile"
#elif SYSTEM == ’b’
    #include "uid.infile"
#endif
```

If the target system is known beforehand, it may be possible to write a program in this way. It may also be possible to design the naming rule of file names of the target system so as to eliminate the necessity of rewriting of the file name.

NAME

C-2-4

CLASSIFICATION

Identifiers, File Names and Header Names

TITLE

Library function (multiple opens of a file)

CLAUSE

7.9.3 Files

GUIDANCE

The same file should not be opened multiple times.

EXPLANATION

Whether the same file can be simultaneously opened multiple times is implementation-defined. An application program should not be coded on the assumption that a file can be opened multiple times.

EXAMPLE

Application Program Without Portability

```
fp1=fopen("spirit.dat","r");
fp2=fopen("spirit.dat","r");
    ...
fscanf(fp1,"X=%d",x);
fscanf(fp2,"Y=%d",y);
```

Application Program With Portability

The following is an alternative:

```
fp1=fopen("spirit.dat","r");
fp2=fopen("tmp.dat","w");
    /* copy the contents from fp1 to fp2 */
fclose(fp1);
fclose(fp2);
fp1=fopen("spirit.dat","r");
fp2=fopen("tmp.dat","r");
    ...
fscanf(fp1,"X=%d",x);
fscanf(fp2,"Y=%d",y);
```

NAME

C-2-5

CLASSIFICATION

Identifiers, File Names and Header Names

TITLE

Library function (*remove*( ) without *fclose*( ))

CLAUSE

7.9.4.1 The *remove*( ) Function

GUIDANCE

An open file must be closed before a *remove*( ) function is executed for it.

EXPLANATION

If the file is open, the behaviour of the *remove*( ) function is implementation-defined. This does not normally happen in a correct program. The *remove*( ) function should be executed on the file after it is closed.

EXAMPLE

Application Program Without Portability

```
fileptr=fopen("fname","r");
    ...
remove("fname");
```

The *remove*( ) function can be processed improperly depending upon the implementation.

Application Program With Portability

```
fileptr=fopen("fname","r");
    ...
fclose(fileptr);
remove("fname");
```

An *fclose*( ) function should be called prior to the execution of the *remove*( ) function.

NAME

C-3-1

CLASSIFICATION

Internal Representation and Type

TITLE

Characters (the range of value of **char** type)

CLAUSE

6.2.1.1 Characters and Integers

GUIDANCE

When **char** type data is used as a value, it should be clearly specified **signed** or **unsigned**.

EXPLANATION

Whether a ‘‘plain’’ **char** has the range of equal values of a **signed char** or of an **unsigned char** is implementation-defined.

EXAMPLE

Application Program Without Portability

```
char a;
    ...
if (a == 255)
    func (a);
```

In the implementation where the ‘‘plain’’ **char** is converted to a **signed int**, the **if** expression will never be satisfied and the *func*( ) function will not be called.

Application Program With Portability

```
unsigned char a;
    ...
if(a == 255)
    func (a);
```

If there are cases where the value of **a** exceeds the range of the **signed char**, as in this case, it should be declared as an **unsigned char**.

NAME

C-3-2

CLASSIFICATION

Internal Representation and Type

TITLE

Floating-point number (truncation at type conversion)

CLAUSE

6.2.1.3 Floating and Integral

GUIDANCE

When the conversion of a value of integer type to floating type causes a truncation, a direction of truncation should not be assumed.

EXPLANATION

When a value of integer type is converted to floating type, if the number of significant digits of the mantissa of the floating-point number is less than that of the integer, cancellation is performed. It is implementation-defined whether the digits cancelled are to be discarded or rounded. For example, if the five digits integer, 12345, is to be converted to a floating-point number which has four significant digits, there are some cases as shown below:

integer	floating-point number	
12345	1.234 X 104	. . . truncation
12345	1.235 X 104	. . . rounding

The program should be coded in such a way so as not to be affected by the difference of precision after the type conversion.

NAME

C-3-3

CLASSIFICATION

Internal Representation and Type

TITLE

Floating-point number (the result when demoted)

CLAUSE

6.2.1.4 Floating Types

GUIDANCE

When a floating-point number is converted to a narrower floating-point number, the direction of truncation or rounding should not be assumed.

EXPLANATION

When a floating-point number is demoted to a narrower floating-point number, the number of significant digits of the mantissa is reduced and it results in cancellation. It is implementation-defined whether such digits are discarded or rounded.

```
float  x;
double y;
x=y;
```

The result of this substitution differs in accordance with the implementation. Therefore, programs should be coded so as not to be affected by the difference of a type conversion or not to accumulate the differences of type conversions.

EXAMPLE

Application Program Without Portability

```
float x;
    ...
x = 10.0;
while (x < 15.0) {
    printf ("x = %e, f(x) = %e0,x,f(x));
    x += 1e-5 ;  /* operation on float */
}
```

Application Program With Portability

```
float x;
long n;
    ...
x=10.0;
n=(15.0-x)/ 1e-5
while (n--> 0){
    printf("x = %e, f(x) = %e0,x,f(x));
    x += 1e-5;  /* operation on double */
}
```

The variance of the number of loops caused by the accumulation of errors may be avoided by counting the number of loops beforehand. The program of this example is portable because it has a loop-counter **n**. However, the errors of the value of the variable **x** still depend on the implementation.

NAME

C-3-4

CLASSIFICATION

Internal Representation and Type

TITLE

Array and pointer (**size_t** type)

CLAUSE

6.3.3.4 The *sizeof* Operator

GUIDANCE

An application program should not assume the type of **size_t**.

EXPLANATION

size_t is an integer type necessary for holding the maximum size of an array. It is defined as an **unsigned** integral type, and the definition of actual type is implementation-defined, such as **unsigned long**, **unsigned int**, and so on.

EXAMPLE

Application Program Without Portability

```
double fa [10000];
unsigned int  x;
x = sizeof fa;
```

When the result of the application of a **sizeof** operator to an array is demoted to **int**, it may result in cancellation.

Application Program With Portability

```
double fa [10000];
size_t  x;
x = sizeof fa;
```

Declare **x** in the form of **size_t** as shown above.

NAME

C-3-5

CLASSIFICATION

Internal Representation and Type

TITLE

Array and pointer (result of type conversion between pointer and integer)

CLAUSE

6.3.4 Cast Operators

GUIDANCE

A pointer may be converted to an integer and back again after the operation. Care must be taken if a type conversion between a pointer and an integer is involved, because the result may vary with the hardware architecture and the value may sometimes not be held.

EXPLANATION

A type conversion between a pointer and an integer is implementation-defined. Some hardware requires a reversion of the upper and lower addresses when addressing a pointer. Application programs written for this type of hardware must be rewritten when they are transferred to hardware which requires no such reversion.

EXAMPLE

Application Program Without Portability

```
char *p,a;
p = &a;
p = (char *) ((int) p+2);
```

The coding shown above is applicable for the hardware in which the result of an operation of an integer value can be used as the address value directly. However, the coding must be rewritten if the hardware requires the different number of bytes to be added when the address is incremented by one byte. An application program with portability may be coded using ‘‘void *’’ as shown below, provided that no operation is performed on a pointer, and only a temporary avoidance of the pointer is required.

Application Program Without Portability

```
float *fp[F_MAX];  /* fp[] and ip[] are to be stacking the pointer */
int   *ip[I_MAX];  /* group for float and the pointer group for int, */
                   /* respectively.*/
struct stack{
    int s_type;    /* stack location of pointer type voided */
    long s_ptr;    /* void locations of various pointers --> */
                   /* using long type */
}s[S_MAX];
i = 0;
s[i].s_ptr = (long)fp[j];  /* The pointer value will not be always */
s[i].s_type=FLOAT;         /* the same as the original value when */
i ++ ;                     /* reconverted, because the pointer is */
                           /* converted to long.*/
j ++ ;
s[i].s_ptr = (long)ip[k];
s[i].s_type = INT;
```

```
i ++ ;
k ++;
    ...
i=0;               /* Following is a loop to return the pointer */
while (... /* continue condition */ ){
    switch(s[i].s_type){
        case INT;
            ip[j++] = (int *) s[i].s_ptr;
                   /* The pointer value will not be always the */
            break; /* same as the original value when reconverted, */
                   /* because the pointer is converted to long.*/
        case FLOAT;
            fp[k++] = (float *)) s[i].s_ptr;
        break;
        ...
    }
    i++ ;
}
```

Application Program With Portability

```
float *fp[F_MAX];  /* fp[] and ip[] are to be stacking the pointer */
int *ip[I_MAX];    /* group for float and the pointer group for int, */
                   /* respectively. */
struct stack{
    int s_type;    /* stack location of pointer type voided */
    void s_ptr;    /* void locations of various pointers --> */
                   /* use 'void *' type */
} s[S_MAX];
i = 0;
s[i].s_ptr = (void *)fp[j];
                   /* If a pointer is converted to 'void *' type, */
s[i].s_type = FLOAT; /* it contains the original value when
                        reconverted. */
i ++ ;
j ++ ;
s[i].s_ptr = (void *)ip[k];
s[i].s_type = INT;
i ++ ;
k ++ ;
i = 0;             /* the following is the loop which returns the
                      pointer. */
while (.. /* continue condition */){
    switch(s[i].s_type){
        case INT;
            ip[j++] = (int *) s[i].s_ptr;
                   /* It is guaranteed that the pointer value is the */
            break; /* same as the original value when reconverted, */
                   /* because it has been converted to */
                   /* 'void *' type. */
        case FLOAT;
            fp[k++] = (float *) s[i].s_ptr;
        break;
        ...
    }
    i++ ;
}
```

NAME

C-3-6

CLASSIFICATION

Internal Representation and Type

TITLE

Array and pointer (**ptrdiff_t** type)

CLAUSE

6.3.6 Additive Operators

GUIDANCE

An application program which assumes the type of **ptrdiff_t** should not be coded.

EXPLANATION

The type of **ptrdiff_t** is a **signed** integral type, and the actual type, such as **long int**, is implementation-defined. Therefore, application programs which are affected by type are not portable.

EXAMPLE

Application Program Without Portability

```
int a[n];
int i,j,k;
i = &a[j] - &a[k];
```

The type of pointer subtraction is **ptrdiff_t**, so it is possible that the variable **i** of **int type** cannot hold the result.

Application Program With Portability

```
int a[n];
ptrdiff_t i;
int j,k;
i = &a[j] - &a[k];
```

NAME

C-3-7

CLASSIFICATION

Internal Representation and Type

TITLE

Structures, unions, enumerations and bit-fields (accessed by a member of a different type)

CLAUSE

6.3.2.3 Structure and Union Members

GUIDANCE

A member of a union object should not be accessed using a member of a different type.

EXPLANATION

If a member of an object is accessed by a member of a different type, the result depends on the internal representation of the data. The internal representation of data is implementation-defined, and therefore programs which are affected by the internal representation of data should not be coded.

EXAMPLE

Application Program Without Portability

```
union { float a;
        struct  { char b;
                  char c[3]; } d
      } e ;
      ...
      /* A process where b is an exponent part, and c is a mantissa.*/
```

The program which depends on the internal representation of data as shown above is not portable.

NAME

C-3-8

CLASSIFICATION

Internal Representation and Type

TITLE

Structures, unions, enumerations and bit-fields (boundary alignment and padding)

CLAUSE

6.5.2.1 Structure and Union Specifiers

GUIDANCE

Application programs should not be coded in a way that assumes the existence or nonexistence of boundary alignment when any member of a structure is accessed.

EXPLANATION

It is implementation-defined whether each member of a structure is to be boundary aligned or not. If a program is coded in such a way that assumes the existence or nonexistence of boundary alignment, it is not portable.

EXAMPLE

Application Program Without Portability

```
struct t { char a;
           int  b; }  x[10][10];
   ...
memcpy (& x[i], & x[j], 80);
```

In the example shown above, an alignment at the 4-byte boundary is assumed. However, the element length of **x** is implementation-defined.

Application Program With Portability

```
struct t { char a;
           int  b;  }  x[10][10];
   ...
memcpy (& x[i], & x[j], sizeof (struct t)*10);
```

NAME

C-3-9

CLASSIFICATION

Internal Representation and Type

TITLE

Structures, unions, enumerations and bit-fields (bit-fields of **int**)

CLAUSE

6.5.2.1 Structure and Union Specifiers

GUIDANCE

In the bit-field of a structure of **int**, whether it is **signed** or **unsigned** must be specified explicitly.

EXPLANATION

Whether a ‘‘plain’’ **int** bit-field is considered **signed** or **unsigned** is implementation-defined. In order to keep the application program portable, either **signed** or **unsigned** must be indicated.

EXAMPLE

Application Program Without Portability

```
struct t { int a:4 ;
           int b:4 ; } x;
x.a = 15;
```

The range of the values which may be entered in **x.a** depends on whether the **int** type indicates **signed** or **unsigned**. If **a** may have a value of 0 through 15, it should be coded as **unsigned int a:4**.

Application Program With Portability

```
struct t { unsigned int a:4;
           unsigned int b:4; } x;
x.a = 15;
```

NAME

C-3-10

CLASSIFICATION

Internal Representation and Type

TITLE

Structures, unions, enumerations and bit-fields (order of allocation of bit-fields)

CLAUSE

6.5.2.1 Structure and Union Specifiers

GUIDANCE

The order of allocation of bit-fields is implementation-defined. Therefore, a program which assumes the order of allocation of bit-fields should not be coded.

EXPLANATION

The order of allocation of bit-fields within a unit is implementation-defined, and usually it is from high-order to low-order, but in some implementations, such as 80x86, it is from low-order to high-order. This presents no problem to the telecommunication protocol process, which mostly manipulates bit data, because it uses a routine only for the purpose of bit handling, and file exchange is done on a character basis. As the order of allocation of bit-fields varies with implementations, the data exchange which is sensitive to bit is not portable.

EXAMPLE

Application Program Without Portability

```
struct tag {
             unsigned a:3,
                      b:4,
                      c:25;
};
union utag {
             struct tag x;
             unsigned   m;
} u;
F()   {
        u.x.a = 0x6;
         u.x.b = 0xf;
         u.x.c = 0x1ffffff;
         if (u.m == 0xdfffffff)
         {
         /* Execute this if it is from the highest-order bit */
         }
         else {
         /* Execute this if it is from the lowest-order bit */
         }
}
```

NAME

C-3-11

CLASSIFICATION

Internal Representation and Type

TITLE

Structures, unions, enumerations and bit-fields (boundary of bit-fields)

CLAUSE

6.5.2.1 Structure and Union Specifiers

GUIDANCE

Programs should not be coded in such a way that a bit-field straddles a storage-unit boundary.

EXPLANATION

It is implementation-defined whether to allocate storage-units straddling a boundary or to allocate the next unit when a bit-field which is to straddle a storage-unit boundary is declared.

EXAMPLE

Application Program Without Portability

```
struct tag { unsigned int  a:16;
             unsigned int  b:24; } x;
p = (int *) &x;      /* Assuming that b is in the field immediately */
if ((*p) == 0) ...   /* following a */
```

Application Program With Portability

```
struct tag { unsigned int  a:16;
             unsigned int  b1:16;
             unsigned int  b2:8; } x;
p = (int *) &x;    /* As b is in the field immediately following a, */
if ((*p) == 0) ... /* so that b1 works as expected. */
```

NAME

C-3-12

CLASSIFICATION

Internal Representation and Type

TITLE

Representation of floating type

CLAUSE

6.1.2.5 Types

GUIDANCE

Programs should not be coded to assume the internal representation of specific hardware.

EXPLANATION

The internal representation of floating type data is unspecified, and it depends on the hardware. A program which assumes the internal representation of a specific machine type is not portable.

NAME

C-3-13

CLASSIFICATION

Internal Representation and Type

TITLE

Alignment of bit-fields

CLAUSE

6.5.2.1 Structure and Union Specifiers

GUIDANCE

An application program which assumes that bit-fields have a specific boundary alignment should not be coded.

EXPLANATION

Except a special case where the immediately following bit-field of a bit-field with a width of 0 is the boundary of **int**, the alignment of the storage unit allocated with bit-fields is unspecified. Therefore, the application program which assumes that bit-fields have a specific boundary alignment is not portable.

EXAMPLE

Application Program Without Portability

```
struct { char a;
         int b: 8;
         int c: 24; } d;
struct { char x;
         int y; } *p;
p = &d;
p -> y = 0;
```

Which part of a bit-field is reset to 0 varies with the boundary alignment of **b**.

Application Program With Portability

```
struct { char a;
         union { int y;
                  struct { int b: 8;
                           int c: 24; } z;
               } w;
       } d;
d.w.y = 0;
```

In order to make the application program portable, specify field sharing explicitly in the specification of union.

NAME

C-3-14

CLASSIFICATION

Internal Representation and Type

TITLE

Storage arrangement of parameters

CLAUSE

6.7.1 Function Definitions

GUIDANCE

An application program which assumes the arrangement of parameters in the storage should not be coded.

EXPLANATION

How the parameters are arranged in the storage is unspecified. Therefore, the programs which assume the arrangement of parameters are not portable.

EXAMPLE

Application Program Without Portability

```
f(char a,char b)
    { char c[2];
    ...
        memcpy(c,&a,2);  ...}
```

Copying two bytes of the values of parameters, **a** and **b**, by one *memcpy*( ) function is not always successful, because the arrangement of parameters varies with the implementation.

Application Program With Portability

```
f(char a,char b)
    { char c[2];
    ...
        c[0] = a;
        c[1] = b;  ...}
```

NAME

C-4-1

CLASSIFICATION

Arithmetic Operation

TITLE

Order of evaluation of an expression

CLAUSE

6.3 Expressions

GUIDANCE

An application program which assumes the order of evaluation of an expression should not be coded.

EXPLANATION

Even if the order of evaluation is intended to be specified explicitly, such an order is unspecified, and the compiler may change it freely within the constraints of the priority. Therefore, if the order of evaluation of an expression is important, specify the sequence point explicitly by using a substitution, for example.

EXAMPLE

Application Program Without Portability

```
int a, b, c, d;
d = a + ( b - c );
    /* It cannot be guaranteed that ’b-c’ should be evaluated earlier. */
```

Application Program With Portability

```
int a, b, c, d;
d = b - c;
d += a;
```

NAME

C-4-2

CLASSIFICATION

Arithmetic Operation

TITLE

Timing of generating side effects

CLAUSE

6.3	Expressions
6.3.2.2	Function Calls

GUIDANCE

The timing of a generation of side effects should not be assumed.

EXPLANATION

The timing of generating side effects is unspecified. The timing may vary depending upon the place where it is described, even within the same implementation.

EXAMPLE

Application Program Without Portability

```
#include <stdio.h>
int i = 0;
printf("side effect sample %d : %d 0, i, i++);
```

In the example shown above, the timing when i++ is evaluated varies with the implementation, and the result can vary as follows:

```
side effect sample 0 : 0
side effect sample 0 : 1
side effect sample 1 : 0
side effect sample 1 : 1
```

Application Program With Portability

An application program should be coded as shown below in order not to be affected by the timing of generation of side effects, depending on the result required.

```
#include <stdio.h>
int i = 0;
int j;
j = i;
i++;
printf("side effect sample %d : %d 0, j, i);
```

The following is a display of the example shown above.

```
side effect sample 0 : 1
```

NAME

C-4-3

CLASSIFICATION

Arithmetic Operation

TITLE

Returned values of *strcmp*( ), *strncmp*( ), *wcscmp*( ) and *wcsncmp*( ) functions

CLAUSE

7.11.4.2	The *strcmp*( ) Function
7.11.4.4	The *strncmp*( ) Function
7.15.2.4.1	The *wcscmp*( ) Function
7.15.2.4.3	The *wcsncmp*( ) Function

GUIDANCE

Values other than zero which are returned by the *strcmp*( ) and *strncmp*( ) functions, or the *wcscmp*( ) and *wcsncmp*( ) functions for different character types, should not be used.

EXPLANATION

These functions may compare character strings, but the values described below should not be used, as each of them depends on the character code:

- values other than zero which are returned by *strcmp*( ) and *strncmp*( ) functions for different character types
- values other than zero which are returned by *wcscmp*( ) and *wcsncmp*( ) functions.

EXAMPLE

Application Program With Portability

```
#include <string.h>
char *func(char *c1, char *c2)
{
if (strcmp(c1, c2) == 0)
    return NULL;
if (strcmp(c1, c2) > 0)
    return c1;
if (strcmp(c1, c2) < 0)
    return c2;
    /* If c1 and c2 contain different types of characters, */
    /* either c1 or c2 is returned depending upon the implementation. */
}
```

For example, if the *strcmp*( ) function and:

c = func("A0", "AA");

are used together, the result will be as follows:

c = "AA" for ASCII code implementation
c = "A0" for EBCDIC code implementation

The following chart shows the codes for "**A**" and "**0**" represented by ASCII and EBCDIC codes, respectively:

	"A"	**"0"**
ASCII	0x41	0x30
EBCDIC	0xC1	0xF0

NAME

C-4-4

CLASSIFICATION

Arithmetic Operation

TITLE

Arithmetic conversion when the result of the operation cannot be represented in storage

CLAUSE

6.2.1.3	Floating and Integral
6.2.1.4	Floating Types
6.3	Expressions
6.3.4	Cast Operators

GUIDANCE

If an arithmetic conversion or operation is expected to produce a result which cannot be represented, an algorithm to check the maximum or minimum space provided should be included before the operation is performed.

EXPLANATION

When a value of floating type is converted to an integral type, the value of the integer cannot be represented by the integral type.

When a **double** is demoted to **float** or a **long double** to **double** or **float**, the value being converted is outside the range of values that can be represented.

When a pointer is converted to an integral type, the length of the range is not enough.

When an arithmetic conversion/operation produces a result which cannot be represented as in the cases described above, the behaviour is undefined. The behaviour varies with the implementation as described below:

- An implementation may abort the process and transfer control to the operating system.
- A system which generates signals.
- An implementation may continue processing with no error indication and with the value indeterminate.
- An implementation may convert with arbitrary truncation.

The behaviour after conversion will strongly depend on the implementation. To avoid this, the value of **_MIN or **_MAX of the converting-to type should be checked before the conversion is performed.

EXAMPLE

Application Program Without Portability

```
#include <limits.h>
long a;
double b;
a = b;  /* If the value of b cannot be represented in a, */
        /* the behaviour is undefined. */
```

Application Program With Portability

```
#include <limits.h>
long a;
double b;
if ((LONG_MIN < b) &&(b < LONG_MAX)) /* checks the range */
    a = b;
else {
    ...  /* describes an exception process */
}
```

NAME

C-4-5

CLASSIFICATION

Arithmetic Operation

TITLE

An object between two sequence points

CLAUSE

6.3 Expressions

GUIDANCE

An object should not be modified more than once between two sequence points.

EXPLANATION

As modifying an object more than once between two sequence points is an undefined behaviour, great care must be taken not to do so.

EXAMPLE

Application Program Without Portability

The example shown below modifies one object twice between two sequence points, so it is undefined. The behaviour varies with the implementation.

```
i = ++i + 1;
```

NAME

C-4-6

CLASSIFICATION

Arithmetic Operation

TITLE

Invalid arithmetic operation

CLAUSE

6.3.5 Multiplicative Operators

GUIDANCE

When an invalid arithmetic operation is suspected, an algorithm which checks it beforehand should be included.

EXPLANATION

For division by zero, for example, the behaviour is undefined and may vary with the implementation as described below:

1. An implementation may abort processing and transfer control to the operating system.
2. The system which generates signals.
3. An implementation may continue processing with no error indication and with the values indeterminate.

The behaviour after such an operation will strongly depend on an implementation. To avoid this, a second operand should be checked prior to the division, if the value of the second operand of the operation is likely to be zero.

EXAMPLE

Application Program Without Portability

```
int a;
int b;
int c;
b = a / c;
  /* If c is zero, the behaviour after division depends on the
     implementation. */
```

Application Program With Portability

```
int a;
int b;
int c;
if (c != 0)
b = a / c;
else {
}
```

NAME

C-4-7

CLASSIFICATION

Arithmetic Operation

TITLE

Incompatibility of types of function with prototype

CLAUSE

6.3.2.2 Function Calls

GUIDANCE

If a function has a function prototype, the declared and defined function types must conform.

EXPLANATION

If a function is called with a function prototype and the function is not defined with a compatible type, the behaviour is undefined. This case arises in the following circumstances:

- An integral type is used, and **long** or **short** and an explicit integer type are used in a function definition/call.
- Specifier of **long** or **short** and an explicit integer type are used in a function prototype, but an integral type is used in a function definition/call, or a function type specification is not explicit.
- When translation units are different, and each function prototype of the translation units is not compatible.

EXAMPLE

Application Program Without Portability

```
long int func1(long int);
func1(long i1)
{
}
```

In the example above, the type of *func1*() is **long int** in the function prototype, but it is **int** in the function definition, because the type is not specified explicitly. Whether the ‘‘plain’’ **int** is interpreted as a synonym with **long int** depends on the implementation; some implementations treat this as a translation error, and others do not.

Application Program With Portability

```
long int func1(long int);
long int func1(long i1)
{  /* Make the function type compatible with the prototype */
}
```

NAME

C-4-8

CLASSIFICATION

Arithmetic Operation

TITLE

Shift operation of invalid number of shifts

CLAUSE

6.3.7 Bitwise Shift Operators

GUIDANCE

When an invalid number of shifts is suspected, an algorithm which checks it beforehand should be included.

EXPLANATION

If a negative number or an amount greater than the width in bits of the expression is being shifted, the behaviour is undefined and the result is not guaranteed. Therefore, when the amount being shifted is uncertain and an invalid shift operation may occur, a check should be performed before the shift operation.

EXAMPLE

Application Program Without Portability

```
#include <limits.h>
unsigned int bitPattern;
int shift;
unsigned int result;
result = (bitPattern << shift);
  /* If a negative number or an amount greater than the width in bits */
  /* of the unsigned int is being shifted, the value of the result
  /* is undefined. The behaviour after this depends on the
     implementation. */
```

Application Program With Portability

```
#include <limits.h>
unsigned int bitPattern;
int shift;
unsigned int result;
if (0 <= shift && shift <= (sizeof(bitPattern) * CHAR_BIT))
    result = (bitPattern << shift);
else {
    ... /* describe an exception process */
     }
```

NAME

C-4-9

CLASSIFICATION

Arithmetic Operation

TITLE

Assignment of an overlapping object

CLAUSE

6.3.16.1	Simple Assignment
7.11.2.1	The *memcpy*( ) Function
7.11.2.3	The *strcpy*( ) Function
7.11.2.4	The *strncpy*( ) Function

GUIDANCE

Avoid assignment of a different object which overlaps the domain.

EXPLANATION

If assignment takes place between objects which overlap, the behaviour is undefined. Therefore, this circumstance should be avoided, if possible. A *memmove*( ) function should be used, if it is inevitable. There are fifteen functions which may cause undefined behaviour: *sprintf*( ), *sscanf*( ), *vsprintf*( ), *mbstowcs*( ), *wcstombs*( ), *strcpy*( ), *strncpy*( ), *strcat*( ), *strncat*( ), *strxfrm*( ), *strftime*( ), *wcscpy*( ), *wcsncpy*( ), *wcscat*( ), *wcsncat*( )

EXAMPLE

Application Program Without Portability

```
#include <string.h>
void insertChar(char ch, char *array, short int id)
{
/* By inserting the character ch after the ordinal number id of
   the character array, move the rest of array after id backward.
   For example, a function which displays "result text:abcdef" by
   coding as shown below: */
    char TextArray[1024] = "abcdef";
    insertChar("1", TextArray, 2);
    printf("result text : %s0, TextArray); */
    char *cpl;
    cpl = array + id;
    strcpy(cpl + 1, cpl); /* Moves the array after id backward */
    *cpl = ch;
}
```

In this example, there is a possibility that the result is unpredictable, depending on the specification of *strcpy*( ).

Application Program With Portability

```
#include <string.h>
void insertChar(char ch, char *array, short int id)
{
    char *cp1, *cp2;
    cp1 = array + id;
    cp2 = array + id + 1;
    memmove(cp2, cp1, strlen(cp1) + 1);
    *cp1 = ch;
}
```

NAME

C-4-10

CLASSIFICATION

Arithmetic Operation

TITLE

Use of function which does not return a value

CLAUSE

6.6.6.4 The *return* Statement

GUIDANCE

The function which is to return a value must return a value.

EXPLANATION

If the function which is to return a value returns no value, the behaviour is undefined. This case arises in the circumstance described below:

- A *return* statement when an *if* statement is not satisfied is missing from the selection statement.

EXAMPLE

Application Program Without Portability

```
int func(int i)
{
 if(i >= 0 && i <= 9)
    return 0;
}
```

In the example above, *func*( ) checks whether the value which is given by the argument is one digit (0 through 9) or not. As no *return* statement for a case when the condition is not met is coded, the behaviour is undefined.

Application Program With Portability

```
int func(int i)
{
    if(i >= 0 && i <= 9)
        return 0;
    return -1;
 /* Describes the return value for a case when the condition is not met. */
}
```

NAME

C-4-11

CLASSIFICATION

Arithmetic Operation

TITLE

Result of integer arithmetic functions

CLAUSE

7.10.6 Integer Arithmetic Functions

GUIDANCE

Note that there are some cases where the results of integer arithmetic functions (abs, div, labs, ldiv) cannot be represented.

EXPLANATION

If the result of an integer arithmetic function cannot be represented, the behaviour is undefined. When there is a possibility of producing a result which cannot be represented, such cases must be prevented by, for example, checking the arguments beforehand.

EXAMPLE

Application Program Without Portability

```
#include <stdlib.h>
main()
{
    int i1 = -32768; int i2;
    i2 = abs(i1);
    ...
}
```

If **int** is two bytes long and the implementation uses two's-complement representation for integers, the absolute value is undefined and the behaviour after running this example depends on the implementation.

NAME

C-4-12

CLASSIFICATION

Arithmetic Operation

TITLE

Using object pointers of automatic storage duration outside of block

CLAUSE

6.1.2.4 Storage Durations of Objects

GUIDANCE

An object pointer which has automatic storage duration should not be used outside the block.

EXPLANATION

If the value stored in a pointer which referred to an object with automatic storage duration is used outside the block, the behaviour is undefined, because the automatic storage may be overwritten and used by another block and the value stored in a pointer may not be a value that is expected. Such an operation will occur in one of the patterns shown below:

1. Returns the object pointer of automatic storage duration in a return value.
2. Assigns the object pointer of automatic storage duration to a variable outside the block.
3. Assigns the object pointer of automatic storage duration to a variable inside the block of static storage duration. And the address of the static variable is transferred to outside by means of 1. or 2. above.

NAME

C-4-13

CLASSIFICATION

Arithmetic Operation

TITLE

Invalid reference

CLAUSE

6.3.3.2	Address and Indirection Operators
6.3.6	Additive Operators

GUIDANCE

Invalid references, such as a reference of NULL pointer, should not be used.

EXPLANATION

For an invalid reference, such as a NULL pointer reference, the behaviour is undefined. Care must be taken not to make invalid references as shown below:

- Use of a unary * operator when an invalid value is assigned to a pointer.
- An invalid array reference (a subscript exceeds the range of array).

NAME

C-4-14

CLASSIFICATION

Arithmetic Operation

TITLE

Reference of automatic storage class object after *longjmp*( )

CLAUSE

7.6.2.1 The *longjmp*( ) Function

GUIDANCE

Care must be taken for referencing an automatic storage class object after *longjmp*( ).

EXPLANATION

If an object of automatic storage class does not have volatile-qualified type after a *longjmp*( ), the behaviour is undefined. That is to say, the object does not always have the same value as it did when *longjmp*( ) was called.

NAME

C-5-1

CLASSIFICATION

Character String Handling

TITLE

Value of the character which cannot be represented

CLAUSE

6.1.3.4 Character Constants

GUIDANCE

The value of the characters which cannot be represented should not be used.

EXPLANATION

The value of a special character such as a control code or an escape sequence which cannot be coded in literals is implementation-defined, like other characters. Therefore, an application program which depends on a value of these characters is not portable. Some of the control codes which support escape sequences may work in one implementation but not in another implementation.

EXAMPLE

Application Program Without Portability

```
#include <stdio.h>
    ...
    putchar(’ 10’);
```

Application Program With Portability

```
#include <stdio.h>
    ...
    putchar(’’);
```

NAME

C-5-2

CLASSIFICATION

Character String Handling

TITLE

Value of a character constant of two or more characters

CLAUSE

6.1.3.4 Character Constants

GUIDANCE

A character constant consisting of two or more characters should not be used.

EXPLANATION

In SPIRIT, character constants composed of two or more characters and wide-character constants are implementation-defined. Therefore, a program which uses a character constant composed of two or more characters is not portable.

EXAMPLE

Application Program Without Portability

```
#include <stdio.h>
    ...
    int c;
    ...
    while ((c = getchar()) != EOF) {
        if (c == ’ab’) {
        ...
    }
}
```

NAME

C-5-3

CLASSIFICATION

Character String Handling

TITLE

Modification of string literals

CLAUSE

6.1.4 String Literals

GUIDANCE

String literals should not be modified.

EXPLANATION

If a program attempts to modify a string literal, the behaviour is undefined. In SPIRIT C, string literals whose appearance is the same need not be stored separately. Care must be taken, because some implementations allocate a memory area for storing string literals where writing is prohibited by the system.

EXAMPLE

The first example shown below is not portable, because the string literal is being modified through a pointer. The second example may seem the same as the first one, but it is not a modification of a string literal, because it is defined that the string which is written as the initialised data of a **char** type array is interpreted as follows:

```
static char string[] =
    [’N’,’u’,’m’,’b’,’e’,’r’,’ ’,’x’,’ ’];
```

Application Program Without Portability

```
char *string = "Number x";
...
string[7] = ’1’;
```

Application Program With Portability

```
static char string [] = "Number x";
...
string[7] = ’1’;
```

NAME

C-5-4

CLASSIFICATION

Character String Handling

TITLE

Contiguity with wide-character string literals

CLAUSE

6.1.4 String Literals

GUIDANCE

Character string literals should not be placed adjacent to wide-character string literals, and *vice versa*.

EXPLANATION

In SPIRIT, character string literals and wide-character string literals are combined into one string if they are adjacent to each other in a source file. Therefore, it is no problem when two character string literals or two wide-character string literals are adjacent to each other. However, when a character string literal is adjacent to a wide-character string literal, the behaviour is undefined, because it is general that the internal representation of a character string literal and a wide-character string literal is different.

EXAMPLE

Application Program Without Portability

```
strp = "Please don’t put string literals and wide-character string
        literals0

"next to each other in your source code. 0

L" そのようなプログラムには移植性がありません . \n";
```

NAME

C-5-5

CLASSIFICATION

Character String Handling

TITLE

Argument of character handling function outside of domain

CLAUSE

7.3 Character Handling **<ctype.h>**

GUIDANCE

Check the domain prior to invocation of a function.

EXPLANATION

All the arguments of character handling functions, such as the *isalpha*( ) function which tests the character set, must have **int** type. In SPIRIT, when such an argument is given a value outside of the range which can be represented in **unsigned char** type, or a value other than EOF, the result is undefined.

EXAMPLE

Application Program Without Portability

In the example shown below, the result of execution is not guaranteed, because the value of **c** (generally speaking) exceeds the range of value that an **unsigned char** can have.

```
#include <ctype.h>
    ...
    int c;
    ...
    c = 1000;
    if (isalpha(c)) {
    ...
    }
```

Application Program With Portability

As the first example is not practical, another example is shown below. If **c** is the value returned by the function, like a *getchar*( ) function, which returns a value satisfying the domain check, this checking is not necessary.

```
#include <limits.h>
#include <ctype.h>
    ...
int c;
    ...
    if ((c >= 0 && c <= UCHAR_MAX) || c == EOF) {
    if (isalpha(c)) {
    ...
    }
}
```

NAME

C-5-6

CLASSIFICATION

Character String Handling

TITLE

Changing LC_CTYPE category of locale

CLAUSE

7.10.7 Multibyte Character Functions

GUIDANCE

When LC_CTYPE is changed, initialise the shift state.

EXPLANATION

It is undefined whether the shift states for the multi-byte character functions, such as *mblen*( ), *mbtowc*( ), *wctomb*( ), and so on, are reset to the initial state when the LC_CTYPE category of the current locale is changed. Therefore, the application program which may possibly change the LC_CTYPE category during processing multi-byte character strings is not portable. When the LC_CTYPE category is changed, the shift state must be initialised by passing a null pointer to the character pointer argument of these functions.

NAME

C-5-7

CLASSIFICATION

Character String Handling

TITLE

Characters to be checked with functions *isalnum*( ), and so on

CLAUSE

7.3.1 Character Testing Functions

GUIDANCE

Functions defined in Clause 7.3.1 should be used for testing of characters. However, the range of characters for which *iscntrl*( ), *isgraph*( ), *isprint*( ) and *ispunct*( ) are true remains partially implementation-defined in LOCALE. So, these functions should not be used assuming the behaviour exceeding the specification of SPIRIT C described below.

EXPLANATION

In the SPIRIT C specification, it is defined that *iscntrl*( ) is true for ‘‘\n’’, ‘‘\t’’, ‘‘\v’, ‘‘\b’’, ‘‘\f’’, ‘‘\r’’, ‘‘\a’’ or any one of the implementation-defined control characters, and *isgraph*( ) is true for a basic execution character (see Clause 5.2.1) or ’@’ or any one of the implementation-defined characters.

NAME

C-5-8

CLASSIFICATION

Character String Handling

TITLE

Character testing, character type conversion functions

CLAUSE

7.3 Character Handling **<ctype.h>**

GUIDANCE

The implementation-defined features of character testing and character type conversion functions should not be used.

EXPLANATION

In SPIRIT C, the behaviour of the *iscntl*( ) and *isgraph*( ) functions in an implementation-defined LOCALE locale is defined in detail, but the behaviour of implementation-defined features in ANSI C are all implementation-defined. For only nine functions listed below, the same behaviour in any SPIRIT conforming implementation is guaranteed all the time: *isalnum*( ), *isalpha*( ), *isdigit*( ), *islower*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *toupper*( ), *tolower*( ).

NAME

C-5-9

CLASSIFICATION

Character String Handling

TITLE

Collating sequence of the execution character set

CLAUSE

7.11.4.3	The *strcoll*( ) Function
7.11.4.5	The *strxfrm*( ) Function

GUIDANCE

In any locale defined in SPIRIT, the collating sequence of all execution characters for *strcoll*( ) and *strxfrm*( ) is not defined.

EXPLANATION

In SPIRIT, the collating sequence of the execution character set for *strcoll*( ) and *strxfrm*( ) is implementation-defined. Therefore, the application programs in which *strcoll*( ) and *strxfrm*( ) functions are used must be rewritten when they are ported to another system.

NAME

C-5-10

CLASSIFICATION

Character String Handling

TITLE

Native locale (""Locale) in localisation

CLAUSE

7.4.1.1 The *setlocale*( ) Function

GUIDANCE

The ""locale should not be used in localisation, except when the ""locale is equal to a LOCALE locale.

EXPLANATION

The ""locale (native locale) in localisation is used to define the implementation-defined unique environment. Therefore, the application program which uses this locale is generally less portable, except in the implementations where the ""locale is equal to the LOCALE locale.

NAME

C-5-11

CLASSIFICATION

Character String Handling

TITLE

Source character set

CLAUSE

5.2.1 Character Sets

GUIDANCE

In a source code, the characters other than those specified in Section 2.2.1 of ANSI C or the SPIRIT common character set should not be used.

EXPLANATION

In SPIRIT, it is defined that the source and execution character set in LOCALE locale must contain characters of the character set of the ANSI "C" locale and the SPIRIT common character set.

NAME

C-6-1

CLASSIFICATION

Input/Output

TITLE

A new-line character in the last line of text stream

CLAUSE

7.9.2 Streams

GUIDANCE

A new-line character at the final position of the last line should not be assumed, but should be coded.

EXPLANATION

It is implementation-defined whether the last line of a text stream requires a terminating new-line character. The end of data possibly may not be recognised if there is no new-line character at the end of output.

EXAMPLE

Application Program Without Portability

1. An example which reads one line from a data stream:

```
#include <stdio.h>
FILE *stream
char buffer[81];
int i,ch;
    ...
for (i=0; (i<80) && ((ch = getchar()) != ’0); i++)
    buffer[i] = ch;
   /* As it is not guaranteed that the last line ends with a
      new-line character, it is possible for data to be read beyond
      the EOF. */
buffer[i] = ’ ’;
```

2. An example of output of the last line:

```
#include <stdio.h>
FILE *fp;
fprintf(fp, "End-of-output");
fclose(fp);
/* Closing without output of a new-line character, the data in the
   last line will not be guaranteed depending on the implementation. */
```

Application Program With Portability

1. An example which reads one line from a data stream:

```
#include <stdio.h>
FILE *stream;
char buffer[81];
int i,ch;
    ...
for (i=0; (i<80) && ((ch = getchar() != EOF) &&
(ch != ’0); i++)    /* checks EOF */
    buffer[i] = ch;
buffer[i] = ’ ’;
```

2. An example of output of the last line:

```
#include <stdio.h>
FILE *fp;
fprintf(fp, "End-of-output. 0);
fclose(fp);
    /* Closing the file after output of a new-line character, */
    /* the data of the last line is guaranteed. */
```

NAME

C-6-2

CLASSIFICATION

Input/Output

TITLE

Space characters immediately before the new-line character

CLAUSE

7.9.2 Streams

GUIDANCE

Whether or not the space character sequence which is written out immediately before the new-line character appears when it is read from a text stream should not be assumed.

EXPLANATION

It is implementation-defined whether or not the space character sequence which is written out immediately before a new-line character appears when it is read from a text stream.

EXAMPLE

Application Program Without Portability

```
#include <stdio.h>
FILE *stream;
    fprintf(stream, "SPIRIT 0);
    /* A space immediately before a new-line character is not
       guaranteed. */
```

Application Program With Portability

```
#include <stdio.h>
FILE *stream;
    fprintf(stream, "SPIRIT\n");
    /* A space immediately before a new-line character should not
       be coded. */
```

NAME

C-6-3

CLASSIFICATION

Input/Output

TITLE

The number of null characters to be appended

CLAUSE

7.9.2	Streams

GUIDANCE

The number of null characters to be appended to a binary stream should not be assumed.

EXPLANATION

The number of null characters appended to data which is written in a binary stream is implementation-defined.

EXAMPLE

It is supposed that data is written in a stream:

```
#include <stdio.h>
FILE *stream;
int int_1, int_2, ch;
float flt_1, flt_2;
stream = fopen("myfile.dat", "wb");
fwrite(&flt_1, sizeof(float), 1, stream);
fwrite(&int_1, sizeof(int), 1, stream);
fclose(stream);
```

Application Program Without Portability

```
stream = fopen(myfile,dat", "rb");
fseek(stream, (long)(-sizeof(int)), SEEK_END);
    /* Correct positioning is not guaranteed, because there are
       unknown number of null characters before EOF. */
fread(&int_2, sizeof(int),1, stream);
```

Application Program With Portability

```
stream = fopen("myfile.dat", "rb");
fseek(stream, (long)(sizeof(float)), SEEK_SET);
fread(&int_2, sizeof(int),1, stream);
    /* Positioning from the top of the file. */
```

NAME

C-6-4

CLASSIFICATION

Input/Output

TITLE

File position in append mode

CLAUSE

7.9.3 Files

GUIDANCE

A file position indicator should not be referenced immediately after the file is opened with append mode.

EXPLANATION

It is implementation-defined whether the file position indicator is initially positioned at the beginning or end of the file when the file is opened with append mode.

EXAMPLE

The program is supposed to be one that reads the last position of a file.

Application Program Without Portability

```
FILE *stream;
long position;
stream = fopen("myfile.dat", "a");
    /* At this time, there is a fear of the file position indicator
       positioned at the beginning of the file. */
position = ftell(stream);
```

Application Program With Portability

```
FILE *stream;
long position;
stream = fopen("myfile.dat", "a");
fseek(stream, 0L, SEEK_END);
    /* The file position indicator is positioned at the end of the file. */
position = ftell(stream);
```

NAME

C-6-5

CLASSIFICATION

Input/Output

TITLE

Writing into text stream in update mode

CLAUSE

7.9.3 Files

GUIDANCE

When a write on a text stream in update mode is executed, whether or not the subsequent data of the associated file remains should not be assumed.

EXPLANATION

It is implementation-defined whether a write on a text stream in update mode causes the associated file to be truncated beyond that point. When the data subsequent to the point of the stream where data is written should not be truncated, it is common to prepare a work file in addition to the update file and rename it after all the necessary update activities are done on the work file. When frequent partial updating on a file is expected, the file is usually generated using a binary stream.

EXAMPLE

The program is to write all the characters in "myfile.dat" to the "output.dat" after replacing the first character of "myfile.dat" with "!".

Application Program Without Portability

```
#include <stdio.h>
FILE *stream, *stream1;
stream = fopen("myfile.dat","r+");
    /* To be opened with text mode */
fputc('!', stream);
stream1 = fopen("output.dat", "w");
fseek(stream, 0L, SEEK_SET);
while (!feof(stream))
    /* Copying characters except '!' is not guaranteed */
fputc(fgetc(stream), stream1);
```

Application Program With Portability

```
include <stdio.h>
FILE *stream, *stream1;
stream = fopen("myfile.dat", "rb+");
    /* To be opened with binary mode */
fputc('!', stream);
stream1 = fopen("output.dat", "w");
fseek(stream, 0L, SEEK_SET);
while (!feof(stream))   /* All the characters will be copied */
fputc(fgetc(stream), stream1);
```

NAME

C-6-6

CLASSIFICATION

Input/Output

TITLE

File of zero-length

CLAUSE

7.9.3 Files

GUIDANCE

A program which assumes the existence or non-existence of a file of zero length should not be coded.

EXPLANATION

Whether a file of zero length actually exists is implementation-defined.

EXAMPLE

Application Program Without Portability

1. An example which assumes the existence of a file of zero length:

```
#include <stdio.h>
FILE *stream, *stream1;
stream = fopen("myfile.dat", "w");
fclose(stream);
stream1 = fopen("myfile.dat", "r");
/* In the implementation which cannot generate a file of zero length,
    the stream1 is a NULL pointer. */
if ((c = fgetc(stream1)) == EOF) printf("No data 0);
/* fgetc does not behave correctly, because the stream1 can be a
   NULL pointer. */
```

2. An example which assumes the nonexistence of a file of zero length:

```
#include <stdio.h>
FILE *fp;
int w_flg;    /* A flag indicating if the file is written */
char ch, s[L_tmpnam];
tmpnam(s);    /* Open the temporary file, and remove it after
                 completion */
fp = fopen(s, "w");
if (...) {
    fputc(ch, fp);
    w_flg = 1;
}
    ...
fclose(fp);
if (w_flg)
    remove (s);
    /* It is possible that file of zero length may exist */
```

Application Program With Portability

1. An example which assumed the existence of a file of zero length above:

```
#include <stdio.h>
FILE *stream, *stream1;
stream = fopen("myfile.dat", "w");
fclose(stream);
if((stream1 = fopen("myfile.dat", "r")) == NULL)
    /* Checks if fopen results in error */
    printf("No data 0);
else
    if((c = fgetc(stream1)) == EOF) printf("No data 0);
```

2. An example which assumed the nonexistence of a file of zero length above:

```
#include <stdio.h>
FILE *fp;
int w_flg;    /* A flag indicating if a file is written */
char ch, s[L_tmpnam];
tmpnam(s);    /* Open the temporary file, and remove it after
                 completion */
fp = fopen(s, "w");
if (...) {
    fputc(ch, fp);
    w_flg = 1;
}
fclose(fp);
remove (s);   /* Call the remove( ) function regardless of write */
              /* The file is removed regardless of implementation */
```

NAME

C-6-7

CLASSIFICATION

Input/Output

TITLE

Conversion specifier %p for input or output

CLAUSE

7.9.6.1	The *fprintf*( ) Function
7.9.6.2	The *fscanf*( ) Function

GUIDANCE

A program which depends on the value of a pointer of different output format and execution unit should not be coded.

EXPLANATION

The output format of a pointer is implementation-defined. When the execution unit is different, reference to a pointer value in the different unit is not guaranteed. In this case, the behaviour is undefined.

EXAMPLE

In the example, it is assumed that a pointer is written into a file in the following format:

```
#include <stdio.h>
FILE *stream;
int i;
int *ptr;
ptr = &i;
fprintf(stream, "%p", (void *) ptr);
fclose(stream);
```

Application Program Without Portability

1.
```
#include <stdio.h>
FILE *stream;
char st1[P_LEN];
fscanf(stream, "%s", st1);
if (strcmp(st1, "1234:5678")) {
    /* Output format depends on the implementation */
}
```

2. The following is assumed to be in the function belonging to the different execution unit from one where the pointer is written out.

```
#include <stdio.h>
FILE *stream;
int i;
int *ptr1;
fscanf(stream, "%p", (void *) ptr1);
    /* Reference to the pointer in a different execution unit is
       not guaranteed. */
```

NAME

C-6-8

CLASSIFICATION

Input/Output

TITLE

%[of the *fscanf*( ) function

CLAUSE

7.9.6.2 The *fscanf*( ) Function

GUIDANCE

A ’-’ character should not be used except as the first or the last character in the scanlist for %[conversion in the *fscanf*( ) function.

EXPLANATION

Some implementations may interpret ’-’ as a specifier of a range, and others as a simple character ’-’. The choice is implementation-defined.

EXAMPLE

Application Program Without Portability

```
#include <stdio.h>
FILE *stream,
char st[5];
fscanf(stream, "%[0-3]",st);
    /* This identifies either a sequence of "0123" or a combination
       of characters ’0’, ’-’ and ’3’, according to circumstances. */
```

Application Program With Portability

If "0123" is desired, it should be coded as shown below:

```
#include <stdio.h>
FILE *stream,
char st[5];
fscanf(stream, "%[0123]", st);
Otherwise, use "%[-03]" or "%[03-]".
```

NAME

C-6-9

CLASSIFICATION

Input/Output

TITLE

Abort with opened files and temporary files

CLAUSE

7.9.4.3	The *tmpfile*( ) Function
7.10.4.1	The *abort*( ) Function

GUIDANCE

Files should not be aborted while they are open.

EXPLANATION

When abort occurs while files are open or temporary files are not deleted, it is implementation-defined whether they are closed/deleted or not.

EXAMPLE

Application Program Without Portability

```
#include <stdio.h>
FILE *stream;
fopen(stream, "file1", "w+");
abort( );  /* It is not guaranteed that the file will be closed. */
```

Application Program With Portability

```
#include <stdio.h>
FILE  *stream;
fopen(stream, "file1", "w+");
fclose(stream);
abort( );  /* The program is aborted after the file is closed. */
```

NAME

C-6-10

CLASSIFICATION

Input/Output

TITLE

Implementation-dependent character printing operation

CLAUSE

5.2.2 Character Display Semantics

GUIDANCE

Printing as specified below should not be done:

- printing a character at the final position of a line
- printing a backspace character at the initial position of a line
- printing a horizontal-tab character at or past the last defined horizontal tabulation position
- printing a vertical-tab character at or past the last defined vertical tabulation position.

EXPLANATION

The behaviours for the above circumstances are unspecified.

NAME

C-6-11

CLASSIFICATION

Input/Output

TITLE

File position of the *ungetc*( ) function

CLAUSE

7.9.7.11 The *ungetc*( ) Function

GUIDANCE

The file position indicator should not be referenced until all the pushed-back characters are read or discarded.

EXPLANATION

The value of the file position indicator until all the pushed-back characters are read or discarded after the successful call to an *ungetc*() function for the text stream varies with the implementation. The behaviour is unspecified.

EXAMPLE

Application Program Without Portability

```
#include <stdio.h>
#include <ctype.h>
FILE *stream;
int ch;
unsigned int result = 0;
long fp1;
while ((ch = getc(stream)) != EOF && isdigit(ch))
    result = result * 10 + ch - ’0’;
if (ch != EOF)   {
    ungetc(ch, stream);
    fp1 = ftell(stream);
    /* The file position indicator will not be referenced until
       the pushed-back characters by the ungetc() function are read
       or discarded. */
}
```

Application Program With Portability

```
#include <stdio.h>
#include <ctype.h>
FILE *stream;
int ch;
unsigned int result = 0;
long fptemp, fp1;
while ((ch = getc(stream)) != EOF && isdigit(ch)){
    result = result * 10 + ch - ’0’;
    fptemp = ftell(stream);
}
if (ch != EOF) {
    ungetc(ch, stream);
```

```
        fp1 = fptemp;
        /* The value of the file position indicator of the pushed-back
           character is read before ungetc() is called. */
    }
```

NAME

C-6-12

CLASSIFICATION

Input/Output

TITLE

Details of values of the *fgetpos*( ) and *ftell*( ) functions

CLAUSE

7.9.8.3	The *fgetpos*( ) Function
7.9.8.6	The *ftell*( ) Function

GUIDANCE

The value of the file position indicator should not be referenced directly.

EXPLANATION

The value stored by the *fgetpos*( ) function and the value stored in a text stream by the *ftell*( ) function are strongly dependent on the implementation. The behaviour is unspecified.

EXAMPLE

Application Program Without Portability

```
#include <stdio.h>
FILE *fp;
long tel1, tel2;
fp = fopen("SPIRIT", "r");
tel1 = ftell(fp);
while(fgetc(fp) != ’0)
tel2 = ftell(fp);
printf("Number of char =%1d 0, tel2 - tel1);
    /* This is to output the character count of the text stream using
       the returned value of the ftell() function.
       However, the meaning of the value is not defined in the
       text stream. */
```

Application Program With Portability

```
#include <stdio.h>
FILE *fp;
long tel1,tel2;
int cnt;
fp = fopen("SPIRIT", "r");
cnt = 0;
while (fgetc(fp) != "0)
    cnt ++;
printf("Number of char =%1d 0,cnt);
    /* Count the number of fgetc() function calls */
```

NAME

C-7-1

CLASSIFICATION

Error, Diagnostics

TITLE

Library function (treatment at the time of domain error)

CLAUSE

7.2 Diagnostics **<assert.h>**

GUIDANCE

Domain error should be determined by checking whether or not EDOM is stored in *errno*.

EXPLANATION

Values which mathematical functions return are implementation-defined.

EXAMPLE

Application Program Without Portability

This program assumes that a negative value is returned when domain error occurs in the *sqrt*() function.

```
#include <math.h>
#include <stdio.h>
double x, y;
if ((y = sqrt(x)) < 0
/* The value of a mathematical function at domain error is
   implementation-defined */
    printf("Domain Error0);
```

Application Program With Portability

```
#include <math.h>
#include <errno.h>
#include <stdio.h>
double x, y;
y = sqrt(x);
if (errno == EDOM)
    /* domain error should be determined based on a value of errno */
    printf("Domain Error0);
```

NAME

C-7-2

CLASSIFICATION

Error, Diagnostics

TITLE

errno at the time of underflow

CLAUSE

7.5.1 Treatment of Error Conditions

GUIDANCE

It should not be assumed that ERANGE is set at the time of underflow.

EXPLANATION

Whether or not mathematical functions set the value of the macro ERANGE in the integer expression *errno* at the time of range error of underflow is implementation-defined.

EXAMPLE

Application Program Without Portability

```
#include <math.h>
#include <errno.h>
#include <stdio.h>
double x, y;
    /* x contains a very small value */
y = log10(x);
if (errno == ERANGE)  /* whether or not ERANGE is set at the time of
                         underflow is implementation-defined */
    printf("Underflow 0);
```

NAME

C-7-3

CLASSIFICATION

Error, Diagnostics

TITLE

errno set by the *fgetpos*( ) and *ftell*( ) functions

CLAUSE

7.9.8.3	The *fgetpos*( ) Function
7.9.8.6	The *ftell*( ) Function

GUIDANCE

The value of *errno* on failure should be determined whether a positive value is set.

EXPLANATION

The value that the *fgetpos*( ) or *ftell*( ) functions set in *errno* on failure is implementation-defined.

EXAMPLE

Application Program Without Portability

```
#include <errno.h>
#include <stdio.h>
if (errno == 1)  /* the value may not be 1 depending on implementation */
    printf("error 0);
```

Application Program With Portability

```
#include <errno.h>
#include <stdio.h>
if (errno > 0)  /* not implementation-defined */
    printf("error 0);
```

NAME

C-7-4

CLASSIFICATION

Error, Diagnostics

TITLE

Messages in the *perror*( ) or *strerror*( ) functions

CLAUSE

7.9.9.4	The *perror*( ) Function
7.11.6.2	The *strerror*( ) Function

GUIDANCE

These functions should only be used for output.

EXPLANATION

The message is implementation-defined.

NAME

C-7-5

CLASSIFICATION

Error, Diagnostics

TITLE

errno

CLAUSE

7.1.4 Errors **<errno.h>**

GUIDANCE

A program should not assume that *errno* is a macro or an external identifier.

EXPLANATION

Whether *errno* is a macro or an external identifier is implementation-dependent. This is an unspecified behaviour.

EXAMPLE

Application Program Without Portability

```
#include <errno.h>
&errno
/* correct when errno is an external identifier, error when errno
   is a macro */
```

NAME

C-8-1

CLASSIFICATION

Environmental Limits

TITLE

Translation limits

CLAUSE

5.2.4.1	Translation Limits
6.1.2	Identifiers
6.5.4	Declarators
6.6.4.2	The *switch* Statement

GUIDANCE

The limits listed in the following table are the least guaranteed values in SPIRIT; therefore, coding exceeding these values should not be made.

EXPLANATION

Refer to the following table.

Items	Limits
Nesting levels of compound statements, iteration control structures and selection control structures.	15
Nesting levels of conditional inclusion.	8
Number of pointers, arrays and function declarators (in any combination) which modify an arithmetic, a structure, a union or incomplete type in a declaration.	12
Nesting levels of parenthesised declarators within a full declarator.	31
Nesting levels of parenthesised expressions within a full expression.	32
Number of the significant initial characters in an internal identifier or a macro name.	31
Number of the significant initial characters in an external identifier.	6
Number of external identifiers in one translation unit.	511
Number of identifiers with block scope declared in one block.	127
Number of macro identifiers simultaneously defined in a translation unit.	1024
Number of parameters in one function definition.	31
Number of arguments in one function call.	31
Number of parameters in one macro definition.	31
Number of arguments in one macro invocation.	31
The maximum number of characters (bytes) in a logical source line.	509
Number of characters in a (concatenated) byte character string literal or wide-character string literal.	509
Number of bytes in an object (in a hosted environment; that is, OS supported execution environment).	32767
Number of nesting levels for the **#include** file.	8
Number of case labels for a *switch* statement (excluding those for any nested *switch* statements).	257
Number of members in a single structure or a union.	127
Number of enumeration constants in a single enumeration.	127
Number of levels of nested structure or union in a single struct-declaration-list.	15

EXAMPLE

An example of the number of case labels in a *switch* statement.

Application Program Without Portability

```
#define CASE1   1
#define CASE300 300
int  sw;
switch(sw) {
    case  CASE1:
        ...  /* 300 case labels against the least guaranteed limit 257 */
    case  CASE300:
}
```

Application Program With Portability

```
#define CASE1   1
#define CASE300 300
int  sw;
switch(sw) {
    case    CASE1    :
    case    CASE257: /* For example, once cut off at 257 labels, */
    default:    {    /* then continue using labels by nesting */
                     /* with default */
    switch(sw)  {
       case    CASE258:
       ...
       case    CASE300:
      }
   }
}
```

NAME

C-8-2

CLASSIFICATION

Environmental Limits

TITLE

Limits at the time of execution

CLAUSE

7.9.4.4	The *tmpnam*( ) Function
7.9.3	Files
7.9.6.1	The *fprintf*( ) Function
7.9.2	Streams
7.10.2.1	The *rand*( ) Function
7.10.4.2	The *atexit*( ) Function

GUIDANCE

The limits listed in the following table are the least guaranteed values in SPIRIT, and coding should be done so as not to exceed these limits at the time of execution.

EXPLANATION

Refer to the following table:

Item	Limit
Maximum times of effective calls, TMP_MAX, for the *tmpnam*( ) function.	25
The maximum number of files FOPEN_MAX which *fopen*( ) can manage simultaneously, including the three standard text streams.	8
The maximum number of bytes which the *fprintf*( ) function can produce by a single conversion.	509
The maximum number of bytes in a line of a text file.	254
Value of buffer size, BUFSIZ, used by the *setbuf*( ) function.	256
The maximum value returned by the *rand*( ) function, RAND_MAX.	32767
The maximum number of functions which can be registered by the *atexit*( ) function.	32

NAME

C-8-3

CLASSIFICATION

Environmental Limits

TITLE

Numerical limits of integral types

CLAUSE

5.2.4.2 Numerical Limits

6.1.2.5 Types

GUIDANCE

The limits listed in the following table are the least guaranteed values in SPIRIT, and coding should be done so as not to exceed these limits at the time of coding as well as of execution. And, at the time of data production, such as the output of a string representing a numerical value corresponding to each type, these values should be fully considered.

EXPLANATION

Refer to the following table:

Items	**Limits**
Maximum number of bits for smallest object except bitfield (=byte), CHAR_BIT.	8
The minimum value for an object of type **signed char**, SCHAR_MIN.	–127
The maximum value for an object of type **signed char**, SCHAR_MAX.	+127
The maximum value for an object of type **unsigned char**, UCHAR_MAX.	255
The minimum value for an object of type char, CHAR_MIN.	(note)
The maximum value for an object of type char, CHAR_MAX.	(note)
Maximum number of bytes in a multi-byte character, MB_LEN_MAX.	2
The minimum value for an object of type **short int**, SHRT_MIN.	–32767
The maximum value for an object of type **short int**, SHRT_MAX.	+32767
The maximum value for an object of type **unsigned short int**, USHRT_MAX.	65535
The minimum value for an object of type **int**, INT_MIN.	–32767
The maximum value for an object of type **int**, INT_MAX.	+32767
The maximum value for an object of type **unsigned int**, UINT_MAX.	65535
The minimum value for an object of type **long int**, LONG_MIN.	–2147483647
The maximum value for an object of type **long int**, LONG_MAX.	+2147483647
The maximum value for an object of type **unsigned long int**, ULONG_MAX.	4294967295

Note: If the value of an object of type **char** is treated as a signed integer when used in an expression, the value of CHAR_MIN shall be the same as that of SCHAR_MIN and the value of CHAR_MAX shall be the same as that of SCHAR_MAX. Otherwise, the value of CHAR_MIN shall be 0 and the value of CHAR_MAX shall be the same that of UCHAR_MAX.

NAME

C-8-4

CLASSIFICATION

Environmental Limits

TITLE

Numerical limits of floating types

CLAUSE

5.2.4.2 Numerical Limits

6.1.2.5 Types

GUIDANCE

The limits listed in the following table are the least guaranteed values in SPIRIT, and coding should be done so as not to exceed these limits at the time of coding as well as of execution. And, at the time of data production, such as the output of a string representing a numerical value corresponding to each type, these values should be fully considered.

EXPLANATION

Refer to the following table:

Item		Limit
Base of exponent representation	FLT_RADIX	2
Number of decimal digits, such that: $\hat{Î}\ (p–1)\ ¥ \log_{10} b\ °\ + \begin{cases} \textit{1: if b is a power of 10} \\ \textit{0: otherwise} \end{cases}$	FLT_DIG DBL_DIG LDBL_DIG	6 14 14
Minimum negative integer such that 10 raised to that power is in the range of normalised floating-point numbers, $È \log_{10} b^{e_{min}-1}$ ˘	FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	–37 –78 –78
Maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\hat{Î} \log_{10}((1–b^{–p})\ ¥\ b^{e_{max}})\ °$	FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38 75 75
Maximum representable finite floating-point number, $(1–b^{–p})\ ¥\ b^{e_{max}}$	FLT_MAX DBL_MAX LDBL_MAX	1E+38 1E+75 1E+75
The difference between 1.0 and the least value greater than 1.0 that is representable in the given floating-point type, (= $b^{1–p}$)	FLT_EPSILON DBL_EPSILON LDBL_EPSILON1E	1E–5 1E–13 1E–13
Minimum normalised positive floating-point number, $b^{e_{min}-1}$	FLT_MIN DBL_MIN LDBL_MIN1E	1E–37 1E–78 1E–78

A normalised floating-point number $x(f_1 > 0$ if $x\ π\ 0)$ is defined by the following model:

$$x = s\ ¥\ b^e\ ¥ \sum_{k=1}^{p} f_k\ ¥\ b^{-k},\ e_{min} \le e \le e_{max}$$

where; s, b, e, p and f_k have the following meanings respectively:

s sign (±1)

b base or radix of exponent representation (an integer>1)

e	exponent (an integer between a minimum e_{min} and a maximum e_{max})
p	precision (the number of base-*b* digits in the significand)
f_k	non-negative integers less than *b* (the significand digits).

NAME

C-9-1

CLASSIFICATION

Obsolescent Feature

TITLE

The placement of a storage-class specifier in the declaration specifiers

CLAUSE

6.9.3 Storage-class Specifiers

GUIDANCE

The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature, which should not be used.

EXPLANATION

The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is accepted by most of the current implementations. But, this is an old format, and it is considered that the format shall not be acceptable in future.

EXAMPLE

In the following declaration, a storage-class declaration in the declaration specifier is placed at the second place.

```
int static i;
```

In order to make it portable, the above declaration should be described as follows:

```
static int i;
```

NAME

C-9-2

CLASSIFICATION

Obsolescent Feature

TITLE

The use of function declarator with empty parentheses

CLAUSE

6.9.4 Function Declarators

GUIDANCE

The function declarator with empty parentheses should not be used because it is an obsolescent feature. Also, the function declarator should not be omitted.

EXPLANATION

The use of the function declarator with empty parentheses is accepted by the current implementations. But this is an old format, and it is considered that the format shall not be acceptable in future. Also, if the function declaration is omitted, it is treated as declared with type **int** in old format; therefore, the same thing is considered in future.

EXAMPLE

The following declaration is an old format, and the information of parameter is not given. In this case, the same treatment as C language before ANSI is made. Also, if a function prototype is fully omitted; it is treated as the following old type function declaration (before ANSI).

```
int f();
```

The above function declaration should be described using the prototype that gives the following information of parameter.

```
/* a case where the first argument is double, and the second is long */
int f(double, long);
```

NAME

C-9-3

CLASSIFICATION

Obsolescent Feature

TITLE

The use of function definitions with separate parameter identifier and Declaration Lists

CLAUSE

6.9.5 Function Definitions

GUIDANCE

This type of definition should not be used, because it is an obsolescent feature.

EXPLANATION

The use of function definitions with separate parameter identifier and declaration lists is accepted by the current implementations. But, this is an old format, and it is considered that the format shall not be acceptable in future.

EXAMPLE

The following function definition is an old format.

```
int fdef(a, b)
double a;
long b;
{ /* function body */ }
```

The above function definition should be rewritten in the following format.

```
int fdef(double a, long b)
{ /* function body */ }
```

NAME

C-9-4

CLASSIFICATION

Obsolescent Feature

TITLE

The use of multiple parameters declared with array type

CLAUSE

6.9.6 Array Parameters

GUIDANCE

The use of multiple parameters declared with an array type in separate lvalues to designate the same object is an obsolescent feature, which should not be used.

EXPLANATION

If multiple parameters have an array type, a compiler does not know whether or not those parameters designate the same object. This becomes a big obstacle for the optimisation of vector operation and parallel processing. In order to utilise the effectiveness of these implementations, multiple array parameters to designate the same object should not be used.

EXAMPLE

Application Program Without Portability

```
main(void)
{
    char a[5];
    f(a,a);
    /* the same argument is given to multiple array parameters */
}
f(char pa[5], char pb[5])
{
    /* Some implementation might process this function assuming
       that the areas designated by pa and pb are not the same
    /* function body */
}
```

NAME

C-10-1

CLASSIFICATION

Undefined Behaviour

TITLE

List of undefined behaviours

CLAUSE

Refer to the following table.

GUIDANCE

The items listed in the following table are undefined behaviours, which are likely to prevent portability of application programs, or cause unexpected problems. This type of programming should be avoided.

EXPLANATION

Refer to the following table:

Classification	Summary of Problems	Clause
Statements and preprocessing directives	Incorrect endings of a source file	5.1.1.2
Character string handling	Invalid multi-byte characters	5.2.1.2
Character string handling	Appearance of an unmatched quotation mark (’ or ")	6.1
Identifiers, etc.	An identifier out of the scope	6.1.2.1
Identifiers, etc.	An identifier that has both internal and external linkage	6.1.2.2
Identifiers, etc.	An identifier with external linkage without external definition	6.7
Internal representation and type	Two declarations to the same object or function that specify incompatible types	6.1.2.6
Character string handling	Unspecified escape sequence (’\’, etc.)	6.1.3.4
Identifiers, etc.	Illegal characters between delimiters of preprocessing tokens	6.1.7
Arithmetic operation	An lvalue with an incomplete type	6.2.2.1
Arithmetic operation	Use of the value of a **void** expression	6.2.2.2
Arithmetic operation	Object access by an incorrect lvalue	6.3
Arithmetic operation	Arguments of a **void** expression	6.3.2.2

Classification	Summary of Problems	Clause
Arithmetic operation	Use of a function pointer after a type conversion	6.3.4
Arithmetic operation	Mutual conversion between a pointer to a function and a pointer to an object	6.3.4
Arithmetic operation	Conversion of a pointer to other than an integral or pointer type	6.3.4
Arithmetic operation	Addition and subtraction of pointers to other than an array	6.3.6
Arithmetic operation	Subtraction between pointers not to point to the same array	6.3.6
Arithmetic operation	Comparison between pointers not to point to the same aggregate or union object	6.3.8
Identifiers, etc.	An identifier for an object with no linkage, and also with an incomplete type	6.5
Identifiers, etc.	A function with block scope declared other than extern	6.5.1
Identifiers, etc.	A bit-field declaration with incorrect type	6.5.2.1
Arithmetic operation	Modification of an object with const-qualified type	6.5.3
Arithmetic operation	Reference to an object with volatile-qualified type	6.5.3
Arithmetic operation	Use of an uninitialised object before a value is assigned	6.5.7
Internal representation and type	Incorrect initialisation of an object with an aggregate or union type	6.5.7
Internal representation and type	A tentative definition of an identifier for an object with internal linkage and an incomplete type	6.7.2
Statements and preprocessing directives	Preprocessing operators **#** which results in an invalid character string literal	6.8.3.2
Statements and preprocessing directives	Concatenation operator **##** which results in an invalid preprocessing token	6.8.3.3
Statements and preprocessing directives	An invalid result after processing of **#line**	6.8.4

Classification	Summary of Problems	Clause
Statements and preprocessing directives	Predefined macro as the subject of **#define** or **#undef**	6.8.8
Statements and preprocessing directives	An invalid place in which a standard header is included	7.1.2
Error, diagnostics	Suppression of the *errno* macro definition	7.1.4
Arithmetic operation	An invalid value of an argument of a library function	7.1.7
Error, diagnostic	Suppression of the *assert* macro definition	7.2
Others	Suppression of the *setjmp* macro definition	7.6
Others	Suppression of the *va_start*, *va_arg*, *va_end* macro definition	7.8.1
Others	An invocation of *va_arg* macro without an actual argument	7.8.1.2
Others	An invocation of *va_end* macro without *va_start* macro	7.8.1.3
Input/Output	Use of the *fflush*( ) function for an input or update stream just after the input operation	7.9.5.2
Input/Output	An invalidity at switching of input/output for an update stream	7.9.5.3
Input/Output	Unmatch of the format and the argument list for *fprintf*( ) or *fscanf*( )	7.9.6
Input/Output	An invalid conversion in a format of *fprintf*( ) or *fscanf*( )	7.9.6
Input/Output	An invalid %% conversion of *fprintf*( ) or *fscanf*( )	7.9.6
Input/Output	An invalid h, l or L conversion specifier for the *fprintf*( ) function	7.9.6.1
Input/Output	An invalid # flag for the *fprintf*( ) function	7.9.6.1
Input/Output	An invalid 0 flag for the *fprintf*( ) function	7.9.6.1
Input/Output	An invalid argument of the *fprintf*( ) function	7.9.6.1
Input/Output	An invalid h, l or L conversion specifier for the *fscanf*( ) function	7.9.6.2

Classification	**Summary of Problems**	**Clause**
Input/Output	The conversion result of the *fscanf*( ) function cannot be represented	7.9.6.2
Character string handling	The conversion result of the *atof*( ), *atoi*( ) or *atol*( ) function cannot be represented	7.10.1
Character string handling	Insufficient size of an array written to by a copying or concatenation function	7.11.27.11.3
Others	An invalid conversion specifier for the *strftime*( ) function	7.12.3.5

NAME

C-11-1

CLASSIFICATION

Others (the periphery of execution environment)

TITLE

Meaning of contents of argument *argv* for the *main*( ) function

CLAUSE

5.1.2.2 Hosted Environment

GUIDANCE

The contents of *argv* are implementation-dependent; therefore, they should be rewritten.

EXPLANATION

The contents and meaning of *argv* are implementation-defined matters; therefore, they should be rewritten.

EXAMPLE

A program to display the contents of command lines.

Application Program Without Portability

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    fputs("command line is ", stdout);
    while(*argv != NULL) {                   .....(1)
        fprintf(stdout, " %s",*argv++);      .....(2)
    }
    fputc('0, stdout);
```

The contents of *argv* are implementation-dependent, and a rewrite may be required. For example, there is a fear that no value is given to *argv*[0] depending on implementation. When NULL is specified to *argv*[0], according to the condition of (1), the statement of (2) is not executed, although the values of and after *arg*[1] have been given.

NAME

C-11-2

CLASSIFICATION

Others (the periphery of execution environment)

TITLE

Return value of the *exit*( ) function

CLAUSE

7.10.4.3 The *exit*( ) Function

GUIDANCE

Values other than 0, EXIT_SUCCESS, EXIT_FAILURE should not be used as return values of the *exit*( ) function.

EXPLANATION

The behaviour when values other than the above three are used is implementation-defined, and generally OS-dependent.

EXAMPLE

A program which returns an error to OS.

Application Program Without Portability

```
exit(1);
```

Application Program With Portability

```
exit(EXIT_FAILURE);
```

Values other than 0, EXIT_SUCCESS, EXIT_FAILURE are OS-dependent, and should not be used.

NAME

C-11-3

CLASSIFICATION

Others (the periphery of execution environment)

TITLE

Environment list and the method for altering the environment list

CLAUSE

7.10.4.4 The *getenv*( ) Function

GUIDANCE

The set of environment names and the method for altering the environment list in the *getenv*( ) function are implementation-defined; therefore, they should not be used.

EXPLANATION

The method for altering the program environment using the *getenv*( ) function is an implementation-defined behaviour, and not portable; therefore, a rewrite is required.

EXAMPLE

A program is to decide whether lower-case letters can be output to a terminal under the condition that lower-case letters may not be used on the terminal when ‘‘UPPER’’ is set into ERR_MESSAGE.

Application Program Without Portability

```
#include <stdlib.h>
#include <string.h>
char *pc;
short msgflg = 0;                                ....(1)
if ((pc = getenv("ERR_MESSAGE")) != NULL {       ....(2)
    if (strcmp(pc,"UPPER") == 0) {               ....(3)
        msgflg = 1;                              ....(4)
    }
}
```

The statement (1) initialises the flag at lower-case letters, and (4) sets the flag at upper-case letters. The environment variable ERR_MESSAGE in (2) and ‘‘UPPER’’ in (3) do not always have the same meaning, respectively, depending on implementations.

NAME

C-11-4

CLASSIFICATION

Others (the periphery of execution environment)

TITLE

Use of the *system*( ) function

CLAUSE

7.10.4.5 The *system*( ) Function

GUIDANCE

The *system*( ) function should not be used.

EXPLANATION

The contents of the **s** character string and the environment of the string to be executed are implementation-defined, and they should not be used.

EXAMPLE

A program that passes arguments "X", "Y", "Z" to a command "abc" and executes a process of "abc".

Application Program Without Portability

```
#include <stdlib.h>
system("abc X Y Z");
```

The *system*( ) function should not be used because it is OS-dependent.

NAME

C-11-5

CLASSIFICATION

Others (the periphery of execution environment)

TITLE

More than one call to the *exit*( ) function

CLAUSE

7.10.4.3 The *exit*( ) Function

GUIDANCE

Attention should be paid so that the *exit*( ) function is not called more than once during program execution.

EXPLANATION

If more than one call to the *exit*( ) function is executed by a program, the behaviour is undefined. The case that the *exit*( ) function itself, or functions which call *exit*( ) function indirectly, are included in functions registered by the *atexit*( ), corresponds to an instant. Such a program is not portable and should be rewritten.

EXAMPLE

Application Program Without Portability

```
#include <stdio.h>
main() {
    void  term(void);
        ...
    if(atexit(term))
        abort();
        ...
    return 0;
}
void  term(void)
{
    fputs("called term()0,stderr);
    exit(EXIT_SUCCESS);
}
```

In this case, a return from *main*( ) is equivalent to calling the *exit*( ) function, therefore, regardless of a return value of the *atexit*( ) function, the *exit*( ) function itself or functions which call the *exit*( ) function indirectly should not be registered by the *atexit*( ) function.

NAME

C-11-6

CLASSIFICATION

Others (Signal)

TITLE

Timing of signal generation

CLAUSE

7.7.1.1 The *signal*( ) Function

GUIDANCE

Be careful that a signal is not always generated. In addition, it is necessary to pay attention to the timing of the generation of signals.

EXPLANATION

A signal may not necessarily occur unless specified by the *raise*() function. And, even if a signal occurs, the behaviour may have an implementation-defined aspect because of the timing of the occurrence. Therefore, rewriting may be done, if necessary.

EXAMPLE

A program which issues a message when an interrupt occurs during wait status of an appropriate interval.

Application Program Without Portability

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#define WAIT 10000
#define OFF 0
#define ON 1
volatile sig_atomic_t intrpt = OFF;
void sigint(int sig) {
    intrpt = ON;
}
main()
{
    unsigned long cnt ;
    signal(SIGINT, sigint);
    for(cnt=0; cnt < WAIT ; cnt++)
        ...
    if (intrpt) {                                 ......(1)
        puts("INTERRUPT OCCURRED");
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

If a signal occurs, the signal is not always captured at (1); a rewrite is required.

Application Program With Portability

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#define WAIT 10000
jmp_buf  env;
void sigint(int sig) {
    longjmp(env,1);
}
main()
{
    unsigned long cnt ;
    if(setjmp(env) != 0) {
        puts("INTERRUPT OCCURRED");
        exit(EXIT_SUCCESS);
    }
    signal(SIGINT, sigint);
    for(cnt=0; cnt < WAIT ; cnt++)
        ;
    exit(EXIT_FAILURE);
}
```

NAME

C-11-7

CLASSIFICATION

Others (Signal)

TITLE

Referring to an object at the time of signal occurrence

CLAUSE

7.7.1.1 The *signal*( ) Function

GUIDANCE

In signal handler, function calls and object references in other than a defined manner should not be done.

EXPLANATION

If the signal occurs other than as the result of calling the *abort*( ) or *raise*( ) functions, and in addition, the signal handler calls any function in the standard library other than the *signal*( ) function itself, or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**, the behaviour is undefined.

EXAMPLE

A program in which the signal handler refers to an object with static storage duration.

Application Program Without Portability

```
#include <signal.h>
#include <setjmp.h>
int error_no;
void memfault(int sig)
{
    error_no = MEMFAULT;
    longjmp(env, 1);
}
```

Application Program With Portability

```
#include <signal.h>
#include <setjmp.h>
volatile sig_atomic_t error_no;
void memfault(int sig)
{
    error_no = MEMFAULT;
    longjmp(env, 1);
}
```

NAME

C-11-8

CLASSIFICATION

Others (Signal)

TITLE

Reference to *errno* at signal occurrence

CLAUSE

7.7.1.1 The *signal*( ) Function

GUIDANCE

If the signal occurs other than as the result of calling the *abort*( ) or *raise*( ) function, *errno* should not be referred to.

EXPLANATION

If a call to the *signal*( ) function results in a *sig_err* return, the value of *errno* is indeterminate.

EXAMPLE

A program which examines an error of the *signal*( ) function.

Application Program Without Portability

```
#include <errno.h>
#include <stdlib.h>
#include <signal.h>
void memfault(int sig)
{
    if (signal(sig, fpe_err) == SIG_ERR) {
        if (errno == 12)
            exit(EXIT_FAILURE);
    }
}
```

The *memfault*( ) function is the signal handler. In this example, when a call to the *signal*( ) function results in a SIG_ERR return, the value of *errno* is not guaranteed; therefore, the coding of the signal handler should recognise this possibility.

NAME

C-11-9

CLASSIFICATION

Others (Signal)

TITLE

The default handling of the *signal*( ) function

CLAUSE

7.7.1.1 The *signal*( ) Function

GUIDANCE

A specific default handling concerning the *signal*( ) function should not be expected.

EXPLANATION

If a specific default handling concerning the *signal*( ) function is expected, it should be explicitly specified in an application program. At program startup, SIG_IGN may be specified for some signals, and SIG_DFL is specified for all other signals. Which signals SIG_IGN is specified for is implementation-defined; therefore, if signals should not be ignored, they should be specified at the beginning of the program.

EXAMPLE

Application Program With Portability

```
#include <signal.h>
void fpe_handl(int);  /* the signal handler of SIGFPE */
main()
{
    signal(SIGABORT, SIG_IGN);
    signal(SIGFPE, fpe_hndl);
    signal(SIGILL, SIG_IGN);
    signal(SIGINT, SIG_IGN);
    signal(SIGSEGV, SIG_IGN);
    signal(SIGTERM, SIG_IGN);
    ...
}
```

NAME

C-11-10

CLASSIFICATION

Others (Signal)

TITLE

Blocking of a signal

CLAUSE

7.7.1.1 The *signal*( ) Function

GUIDANCE

An implementation-defined blocking of a signal should not be used.

EXPLANATION

Blocking of a signal is implementation-defined, therefore, if the firm redefinition is required, it should be explicitly defined in an application program.

EXAMPLE

Application Program With Portability

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
volatile sig_atomic_t fpeerr = OFF;
void sigferr(int sig) {                  ......(1)
    signal(SIGPFE, SIG_IGN);             ......(2)
    fpeerr = ON;
    ...
}
main()
{
    signal(SIGFPE, sigferr);
    ...
if (fpeerr) {
    puts("result value is not correct");
    exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

If SIGFPE occurs during the processing of (1) at the preceding signal occurrence, the behaviour is implementation-defined; therefore, attention is required. At (2), the redefinition has explicitly been done. If a firm redefinition is required, it should be explicitly defined as in *sigferr*.

NAME

C-11-11

CLASSIFICATION

Others (non-local jumps)

TITLE

Context of an invocation of the *setjmp* macro

CLAUSE

7.6.1.1 The *setjmp* Macro

GUIDANCE

If the *setjmp* macro is called, it should not be called from other than the defined context.

EXPLANATION

The contexts in which the *setjmp* macro can appear are defined in the specifications, but, if it is called from other than the defined contexts, the behaviour is undefined; therefore, attention is required. An invocation of the *setjmp* macro shall appear in only one of the following contexts:

- the entire controlling expression of a selection or iteration statement
- one operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement
- the operand of a unary ! operator with the resulting expression being the entire controlling expression of a selection or iteration statement
- the entire expression of an expression statement (possibly cast to void).

EXAMPLE

A program which displays return values of *setjmp*.

Application Program Without Portability

```
#include <setjmp.h>
static int ret_val;
ret_val = setjmp(env)
```

Application Program With Portability

```
#include <setjmp.h>
static int ret_val;
switch (setjmp(env)) {
case 1:
    ret_val = 1;
    break;
case 2:
    ret_val = 2;
    break;
}
```

NAME

C-11-12

CLASSIFICATION

Others (nonlocal jumps)

TITLE

An invocation of the *longjmp*( ) function

CLAUSE

7.6.2.1 The *longjmp*( ) Function

GUIDANCE

The *longjmp*( ) function should not be invoked from a nested signal handler.

EXPLANATION

If the *longjmp*( ) function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behaviour is undefined.

EXAMPLE

Application Program Without Portability

```
#include <setjmp.h>
#include <signal.h>
#define MEMFAULT 1
jmp_buf env:
volatile sig_atomic_t error_no;
signal(SIGFPE, zerodiv);
signal(SIGSEGV, memfault);
void zerodiv(int sig)
{
    ...
/* if SIGSEGV occurs during this duration, memfault() is invoked. */
    ...
}
void memfault(int sig)
{
    error_no = MEMFAULT;
    longjmp(env);                      ......(1)
}
```

The signal handler in which the *longjmp*( ) function is described can be prevented moving to a nested status to some extent by describing a block statement like signal (SIGSEGV,SIG_IGN) at the top of the signal handler. However, it is not the perfect protection because of the difference in timing.

NAME

C-11-13

CLASSIFICATION

Others (nonlocal jumps)

TITLE

An implementation of *setjmp*

CLAUSE

7.6.1.1 The *setjmp* Macro

GUIDANCE

Assuming that *setjmp* is a macro or an external identifier should be avoided.

EXPLANATION

It is unspecified whether *setjmp* is a macro or an external identifier; therefore, program code assuming this should be avoided.

EXAMPLE

A program which gets the location address of the *setjmp*( ) function.

Application Program Without Portability

```
#include <stdio.h>
#include <setjmp.h>
int (*func)();
func = setjmp;                      ......(1)
printf("func address is %p",func);
```

setjmp is a macro or an external identifier depending on implementation; therefore, there is no portability. At (1), there is a possibility of a compilation error depending on the implementation.

NAME

C-11-14

CLASSIFICATION

Others (a variable number of arguments)

TITLE

How to realise *va_end*

CLAUSE

7.8.1.3 The *va_end* Macro

GUIDANCE

Assuming that *va_end* is a macro or an external identifier should be avoided.

EXPLANATION

It is unspecified whether *setjmp* is a macro or an external identifier; therefore, program code assuming this should be avoided.

EXAMPLE

A program to get the location address of the *va_end*( ) function.

Application Program Without Portability

```
#include <stdio.h>
#include <stdarg.h>
int (*func) ();
func = va_end;                            ......(1)
printf("func address is %p",func);
```

As va_end is a macro or an external identifier depending on the implementation, it loses portability to have an address assuming an external identifier. And, depending on the implementation, a compilation error may occur due to statement (1).

NAME

C-11-15

CLASSIFICATION

Others (a variable number of arguments)

TITLE

Definition of a variable number of arguments

CLAUSE

6.7.1 Function Definitions

GUIDANCE

A function which receives a variable number of arguments should be defined using an ellipsis notation.

EXPLANATION

When a function which receives a variable number of arguments is defined without a list of parameters ending with an ellipsis notation, the behaviour is undefined. Then, in such a case, it is necessary to use an ellipsis notation.

EXAMPLE

Application Program Without Portability

```
int f(int i, int j, int k)
    /* intend to receive three arguments and less */
{
    ...
}
main(){
    int i,j,k;
    f(i);
    f(i,j);
    f(i,j,k);
}
```

Application Program With Portability

```
int f(int i,...)   /* intend to receive three arguments and less */
{
    ...
}
main(){
    int i,j,k;
    f(i);
    f(i,j);
    f(i,j,k);
}
```

NAME

C-11-16

CLASSIFICATION

Others (a variable number of arguments)

TITLE

Call of a function with an argument of a *va_list* type object

CLAUSE

7.8 Variable Arguments **<stdarg.h>**

GUIDANCE

When an object **ap** having type *va_list* is passed as an argument to another function, if that function invokes the *va_arg* macro with parameter **ap**, the calling function shall invoke a *va_end* macro with argument **ap** prior to any further reference to **ap**.

EXPLANATION

When an object **ap** having type *va_list* is passed as an argument to another function, if that function invokes the *va_arg* macro with parameter **ap**, the value of **ap** in the calling function is different from the original value prior to the invocation and becomes indeterminate. Then, the calling function shall invoke a *va_end* macro with argument **ap** prior to any further reference to **ap**.

EXAMPLE

Application Program Without Portability

```
#include <stdarg.h>
void f1(char *n_ptr, ...)
{
    va_list ap;
    va_start (ap, n_ptr);
    f2(ap);
    ...
    arg1 = va_arg(ap, int);
    ...
    va_end(ap);
}
- - - - - - - - - - - - - -
#include <stdarg.h>
f2(va_list ap)
{
    int i;
    i = va_arg(ap, int);              ...(1)
    va_arg(ap, int) = 100;            ...(2)
    ...
}
```

Although the value of **ap** is not changed by the invocation of the *va_arg* macro by statement (1), it is changed by statement (2). Then, the value of **ap** in the calling function is indeterminate.

Application Program With Portability

```
#include <stdarg.h>
void f1(char *n_ptr, ...)
{
    va_list ap;
    va_start (ap, n_ptr);
    f2(ap);
    va_end(ap);
    va_start (ap, n_ptr);
    arg1 = va_arg(ap, int);
    ...
    va_end(ap);
}
- - - - - - - - - - - - - - -
#include <stdarg.h>
f2(va_list ap)
{
    int i;
    i = va_arg(ap, int);
    va_arg(ap, int) = 100;
    ...
}
```

NAME

C-11-17

CLASSIFICATION

Others (a variable number of arguments)

TITLE

Parameters of the *va_start* macro

CLAUSE

7.8.1.1 The *va_start* Macro

GUIDANCE

Enough attention should be paid to declare the parameter *parmN* in appropriate type.

EXPLANATION

If the type of the parameter *parmN* of a *va_start* macro is not compatible with the register storage class, with a function or array type, or the type that results after application of the default argument promotions, the behaviour is undefined.

EXAMPLE

Application Program Without Portability

```
#include <stdarg.h>
#define MAXARGS 31
void f1(register int n_ptrs, ...)
{
    va_list ap;
    char *array[MAXARGS];
    va_start(ap, n_ptrs);
    ...
}
```

When the register storage class is specified for the second argument of the *va_start* macro, the behaviour is undefined. So, the argument in appropriate type should be declared.

NAME

C-11-18

CLASSIFICATION

Others (a variable number of arguments)

TITLE

Incompatibility of the type specified as the second parameter of a *va_arg* macro with the type of a variable number of arguments

CLAUSE

7.8.1.2 The *va_arg* Macro

GUIDANCE

The type specified as the second parameter of a *va_arg* macro shall be compatible with the type of the actual next argument in the list of a variable number of arguments; therefore, attention is required.

EXPLANATION

If there is no actual next argument of a *va_arg* macro, or if the type specified as the second parameter of a *va_arg* macro is not compatible with the type of the actual next argument (the type that results after application of the default argument promotions), then the behaviour is undefined. So, ensure that such incompatibility may not occur.

EXAMPLE

Application Program Without Portability

```
void f1 (int n_ptr, ...);
void f2 (void)
{
    int i;
    long l;
    ...
    f1(l, i);
    ...
    f1(2, i, l);  . . . (1)
    ...
}
- - - - - - - - - - - - - -
#include <stdarg.h>
#define MAXARGS 31
void f1 (int n_ptr, ...)
{
    va_list ap;
    int array[MAXARGS];
    int ptr_NO = 0;
    ...
    va_start(ap, n_ptr);
    ...
    while(ptr_NO < n_ptr)
         array[ptr_NO ++] = va_arg(ap, int);
         va_end(ap);
          ...
}
```

This example operates normally for the implementation in which the data size of both **int** type and **long** type is the same. However, for the implementation in which the data size of **int** type and **long** type is not the same, the behaviour is undefined as a result of the invocation of statement (1). Then, it is necessary to rewrite this.

Application Program With Portability

```
void f1 (int n_ptr, ...);
void f2 (void)
{
    int i, j;
    ...
    f1(l, i);
    ...
    f1(2, i, j);
    ...
    }
- - - - - - - - - - - - -
    #include <stdarg.h>
    #define MAXARGS 31
    void f1 (int n_ptr, ...)
    {
        va_list ap;
        int array[MAXARGS];
        int ptr_NO = 0;
        ...
        va_start(ap, n_ptr);
        while(ptr_NO < n_ptr)
           array[ptr_NO ++] = va_arg(ap, int);
           va_end(ap);
           ...
    }
```

NAME

C-11-19

CLASSIFICATION

Others (a variable number of arguments)

TITLE

Return from a function initialised by the *va_start* macro

CLAUSE

7.8.1.3 The *va_end* Macro

GUIDANCE

A *va_end* macro, corresponding to a *va_start* macro should be invoked.

EXPLANATION

If a return from a function whose variable argument list that was initialised by a *va_start* macro occurs before the invocation of a *va_end* macro, the behaviour is undefined. It is necessary to invoke a *va_end* macro, corresponding to a *va_start* macro.

EXAMPLE

A program gathers into an array a list of arguments that are pointers to strings.

Application Program Without Portability

```
#include <stdarg.h>
#define MAXARGS 31
void f1(int n_ptrs, ...)
{
    va_list ap;
    char *array[MAXARGS];
    int ptr_NO = 0;
    if (n_ptrs > MAXARGS)
        n_ptrs = MAXARGS;
    va_start(ap, n_ptrs);
    while(ptr_NO < n_ptrs)
        array[ptr_NO ++] = va_arg(ap, char *);
    f2(n_ptrs, array);
}
```

Application Program With Portability

```
#include <stdarg.h>
#define MAXARGS 31
void f1(int n_ptrs, ...)
{
    va_list ap;
    char *array[MAXARGS];
    int ptr_NO = 0;
    if (n_ptrs > MAXARGS)
        n_ptrs = MAXARGS;
    va_start(ap, n_ptrs);
    while(ptr_NO < n_ptrs)
        array[ptr_NO ++] = va_arg(ap, char *);
    va_end(ap);
    f2(n_ptrs, array);
}
```

NAME

C-11-20

CLASSIFICATION

Others (General utility)

TITLE

The order and contiguity of allocated storage

CLAUSE

7.10.3 Memory Management Functions

GUIDANCE

The order and contiguity of allocated storage should not be assumed.

EXPLANATION

As the order and contiguity of storage allocated by the *calloc*( ), *malloc*( ) and *realloc*( ) functions are unspecified, they should not be assumed.

EXAMPLE

A program which calls the *malloc*( ) functions successively and accesses by an array.

Application Program Without Portability

```
#include <stdlib.h>
int *pi0, *pi1;
pi0 = (int *)malloc(sizeof(int));
pi1 = (int *)malloc(sizeof(int));
pi0[0] = 0;
pi1[0] = 1;
printf("pi0[1] = %d", pi0[1]);              .....(1)
```

If contiguous storage is always allocated by successive calls to the *malloc*( ) function, then ’1’ is always the output from the statement (1). However, if contiguous storage is not always allocated by successive calls to the *malloc*( ) function, then ’1’ is not always the output.

Application Program With Portability

```
#include <stdlib.h>
int *pi0, *pi1;
pi0 = (int *)malloc(sizeof(int)*2);
pi1 = &pi0[1];
pi0[0] = 0;
pi1[0] = 1;
printf("pi0[1] = %d", pi0[1]);
```

When contiguous storage is required, the storage must be allocated at once by the *malloc*( ) function. If required contiguous storage is insufficient during execution of a program, expand the storage using the *realloc*( ) function.

NAME

C-11-21

CLASSIFICATION

Others (General utility)

TITLE

The *bsearch*( ) function when two elements are equal

CLAUSE

7.10.5.1 The *bsearch*( ) Function

GUIDANCE

When the *bsearch*( ) function has two elements which are equal, which element is returned as a result of *bsearch*( ) is unspecified. A program which is dependent on the return of a specific element should not be written.

EXPLANATION

If two elements in an array are equal to the search key, which element is matched is unspecified.

EXAMPLE

A program which counts the number of characters whose values are less than the code of character ’c’ in the specified sorted string.

Application Program Without Portability

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
const char *base[] = {"a","a","b","b","c","c"} ;
const char *key = "c";
main() {
    char *p;
    p = (char *)bsearch((void *)key, (void *) base, sizeof(base),
        sizeof(*base), strcmp);
    if(p == NULL) {
        fprintf(stderr,"not found0);
        exit(0);
            }
printf("count = %d0,(int)(p-base));
    }
```

This program assumes that the string "c" on the left in the array base is the first pointed. However, in some implementations, string "c" on the right may be the first found.

Application Program With Portability

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
const char *base[] = "aabbcc";
const char *key = "c";
main() {
    char *p;
    p = (char *)bsearch((void *)key, (void *) base, sizeof(base),
        sizeof(*base), strcmp);
    if(p == NULL) {
        fprintf(stderr,"not found0);
        exit(0);
        }
    while(p != base)  {      /* When "c" on the right matches first */
        if(*p == *(p-1))
            p--;             /* Pointer is moved to the leftmost "c" */
        else
            break;
        }
    printf("count = %d0,(int)(p-base));
    ...
}
```

NAME

C-11-22

CLASSIFICATION

Others (General utility)

TITLE

The *qsort*( ) function when two elements are equal

CLAUSE

7.10.5.2 The *qsort*( ) Function

GUIDANCE

If two elements are equal, their order in the sorted array is unspecified; therefore, attention is required.

EXPLANATION

The *qsort*( ) function sorts the elements of an array into ascending order. If two elements are equal, their order in the sorted array is unspecified, then a specific order shall not be assumed. A comparison function should be structured so as not to return a value 0 (the result is equal) by being aware of all keys.

NAME

C-11-23

CLASSIFICATION

Others (General utility)

TITLE

The *calloc*( ), *malloc*( ) and *realloc*( ) functions whose sizes are zero

CLAUSE

7.10.3 Memory Management Functions

GUIDANCE

Zero as the argument of the *calloc*( ), *malloc*( ) and *realloc*( ) functions should not be specified.

EXPLANATION

When an argument of the *calloc*( ), *malloc*( ) or *realloc*( ) function is specified as zero, the behaviour is implementation-defined. So, zero should not be specified as the argument.

EXAMPLE

A program to find all the areas where the *malloc*( ) function can operate.

Application Program Without Portability

```
if ((p = malloc(0)) == NULL) {                  ......(1)
    fputs("Cant get malloc area0,stderr);
exit(EXIT_FAILURE);
} else {
}
```

As the behaviour as a result of *malloc*(0) in the statement (1) depends on the implementation, it is necessary to rewrite the statement.

NAME

C-11-24

CLASSIFICATION

Others (General utility)

TITLE

Reference to a pointer to space which has been deallocated by a call to the *free*( ) or *realloc*( ) function

CLAUSE

7.10.3 Memory Management Functions

GUIDANCE

Space which has been deallocated should not be referred to.

EXPLANATION

The value of a pointer to space which has been deallocated is undefined.

EXAMPLE

A program which uses a pointer to space (may possibly be deallocated) which has not been reallocated by the *realloc*( ) function.

Application Program Without Portability

```
#include <stdlib.h>
#define TBLSZ 100
#define ON 1
#define OFF 0
    struct TBL    {
    int flg;
    struct TBL * nextp;
    } ;
typedef struct TBL tblty;
main()   {
    tblty * topp, * curlp, * tmpp;
    int n, m;
    topp = (tblty *)malloc(sizeof(tblty)*TBLSZ);
    curlp = topp;
    ...
    curlp = &topp[n];
    curlp->flg = ON;
    curlp->nextp = &topp[m];
    ...
    if(curlp >= &topp[TBLSZ-1])  {
        topp = (tblty *)realloc(topp, sizeof(tblty):TBLSZ*2);
    }
    ...
    tmpp = curlp->nextp;
                /* nextp is a pointer to the space that has not been */
                /* reallocated yet. Then, the reference is invalid. */
    if(tmmp->flg == ON){
    }
```

A program like this is necessary to reserve sufficient space by the initial *malloc*( ), or to manage a chain to each entry of a table by the number of elements in an array, not by pointers.

NAME

C-11-25

CLASSIFICATION

Others (General utility)

TITLE

Pointer argument of the *free*( ) or the *realloc*( ) function

CLAUSE

7.10.3 Memory Management Functions

GUIDANCE

A pointer to deallocated space as an argument of the *free*( ) or *realloc*( ) function should not be specified. A pointer which does not match a pointer earlier returned by the *calloc*( ), *malloc*( ) or *realloc*( ) function as an argument should not be specified.

EXPLANATION

If a pointer argument of the *free*( ) or *realloc*( ) function does not match a pointer earlier returned by the *calloc*( ), *malloc*( ) or *realloc*( ) function, or if the object pointed to by the pointer has been deallocated by a call to the *free*( ) or *realloc*( ) function, the behaviour is undefined.

EXAMPLE

A program in which an argument of the *free*( ) function does not match a pointer returned by the *malloc*( ) function.

Application Program Without Portability

```
char *pc;
int i;
pc = (char *)malloc(sizeof(char)*5);          ......(1)
for(i = 0; i < 5; i++) {
    *pc++ = ’A’ + i;
    }
free(pc);                                     ......(2)
```

As the pointer returned by statement (1) does not match the pointer passed by statement (2), the behaviour is not guaranteed.

NAME

C-11-26

CLASSIFICATION

Others (General utility)

TITLE

The base time of the *clock*( ) function

CLAUSE

7.12.2.1 The *clock*( ) Function

GUIDANCE

The base time of the *clock*( ) function should not be assumed.

EXPLANATION

As the base time of the *clock*( ) function is implementation-defined, the difference between two calendar times should be used.

EXAMPLE

A program which manages the CPU time.

Application Program Without Portability

```
puts("program start");
if (clock()/CLOCKS_PER_SEC > 10) {
fputs("Time limit !!0,stderr);
exit(EXIT_FAILURE);
}
```

Application Program With Portability

```
clock_t st_clk;
puts("program start");
st_clk = clock();
if (clock()/CLOCKS_PER_SEC > 10) {
fputs("Time limit !!0,stderr);
exit(EXIT_FAILURE);
}
```

As the base time of the *clock*( ) function depends on the implementation, use the difference between two calendar times.

NAME

C-11-27

CLASSIFICATION

Others (General utility)

TITLE

Type of the calendar time returned by the *time*( ) function

CLAUSE

7.12.2.4 The *time*( ) Function

GUIDANCE

A program should not make an assumption based on the value returned by the *time*( ) function.

EXPLANATION

The *time*( ) function returns the implementation's best approximation of the current calendar time and the encoding of the value is unspecified. Therefore, the value of the *time*( ) function should not be assumed.

EXAMPLE

Application Program Without Portability

```
#include <time.h>
main() {
    time_t x,y;
    x = time((time_t *)0);
    while()  {
      if    (x+180 < time((time_t *)0))
        break;
            /* intend to exit from while loop after 3 minutes */
    }
}
```

The value returned by the time function is not always expressed in seconds.

Application Program With Portability

```
#include <time.h>
main() {
    time_t x,y;
    x = time((time_t *)0);
    while()  {
      y = time((time_t *)0);
      if (difftime(y, x) > 180.0)
        break;
            /* intend to exit from while loop after 3 minutes */
    }
}
```

NAME

C-11-28

CLASSIFICATION

Others (General utility)

TITLE

Flag for daylight saving time (DST)

CLAUSE

7.12.1 Components of Time

GUIDANCE

Whether or not Daylight Saving Time is adopted is decided based on the fact that the value of **tm_isdst** is not positive if Daylight Saving Time does not exist.

EXPLANATION

In SPIRIT, it is implementation-defined that the value of **tm_isdst** is zero or negative when DST is not adopted.

EXAMPLE

Application Program Without Portability

```
#include <time.h>
#include <stdio.h>
struct tm *t1;
if(t1->tm_isdst < 0) printf("No DST may exist");
    /* the value of tm_isdst may be zero */
```

Application Program With Portability

```
#include <time.h>
#include <stdio.h>
struct tm *t1
if(t1->tm_isdst <= 0) printf("No DST may exist");
    /* the value of tm_isdst, both zero and negative are examined */
```