Document Number: N2241
Submitter: Keld Simonsen
Submission Date: 2018-04-23
Subject: Reentrant APIs

Summary:
This document is in response to the Danish comments on the NWIP on thread-safe extensions.

It contains some ISO wording because I already had that formulated. We can discuss whether this should be a part of some other document, or a separate document, maybe in the form of a TS.

It addresses issues both contained in POSIX and in ISO/IEC TR 30112 – which is an enhanced locale specification backwards compatible with POSIX and C.

Reentrant APIs that are present in POSIX are backwards compatible. APIs that are not in POSIX, but addresses the enhanced functionality in 30112, could be moved to an informative annex.

The functionality of the APIs are implemented to some extent in GLIBC, but not in their reentrant versions.

The specification is written in a way so that it could be bound to easilly from other programming languages.

INTERNATIONAL STANDARD

ISO/IEC 9899-RAPI WD1

ISO/IEC 9899-RAPI WD1
2018-04-22

Information technology — Programming language C – Reentrant APIs

Technologies de l'information —

This page left for ISO/IEC copyright notices.

Contents **Page**

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

This document was prepared with reference to ISO/IEC 30112 – Specification methods for cultural conventions (under development).

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee 22, *Programming Languages*, their environments and system software interfaces.

Introduction

This document defines C APIs for thread-safe programming via re-entrant APIs.

Information technology — Programming Language C – - re-entant APIS

1 SCOPE

This document specifies reentrant APIs for the C programming language.

The specification is upward compatible with POSIX API specifications. An alignment effort has been undertaken for this specification to be aligned with ISO/IEC 9945. APIs are specified to address handling of facilities of ISO/IEC TR 30112.

2 NORMATIVE REFERENCES

The following referenced documents are indispensible for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 8601, *Data elements and interchange formats - Information interchange - Representation of dates and times*.

ISO/IEC 9899, *Information technology - Programming language C*.

ISO/IEC 9945, *Information technology - Portable Operating System Interface (POSIX)*.

3 TERMS, DEFINITIONS AND NOTATIONS

3.1 Terms and definitions

For the purposes of this document, the following terms and definitions apply. (to be updated)

3.1.1 Bytes and characters

3.1.1.1
byte
An individually addressable unit of data storage that is equal to or larger than an octet, used to store a character or a portion of a character.

Note: A byte is composed of a contiguous sequence of bits, the number of which is implementation defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit

3.1.1.2
character
A member of a set of elements used for the organization, control or representation of data

3.1.1.3
coded character
A sequence of one or more bytes representing a single character

3.1.1.4
text file
A file that contains characters organized into one or more lines

3.1.2 cultural and other major concepts

3.1.2.1
cultural convention (locale)
A data item for information technology that may vary dependent on language, territory, or other cultural habits

3.1.2.2
FDCC
A Formal Definition of a Cultural Convention, that is a cultural convention put into a formal definition scheme

3.1.2.3
FDCC-set
A Set of Formal Definitions of Cultural Conventions (FDCC's). The definition of the subset of a user's information technology environment that depends on language and cultural conventions

Note: the FDCC-set is a superset of the "locale" term in C and POSIX.

3.1.2.4
charmap
A definition of a mapping between symbolic character names and character codes, plus related information

3.1.2.5
repertoiremap
A definition of a mapping between symbolic character names and characters for the repertoire of characters used in a FDCC-set

NOTE: This further described in clause 6.

3.1.3 FDCC categories related

3.1.3.1

character class:
A named set of characters sharing an attribute associated with the name of the class.

3.1.3.2
collation
The logical ordering of strings according to defined precedence rules

3.1.3.3
collating element
The smallest entity used to determine logical ordering

Note: See collating sequence. A collating element consists of either a single character, or two or more characters collating as a single entity. The LC_COLLATE category in the associated FDCC-set determines the set of collating elements.

3.1.3.4
multicharacter collating element
A sequence of two or more characters that collate as an entity

Note: For example, in some languages two characters are sorted as one letter, as in the case for Danish and Norwegian "aa".

3.1.3.5
collating sequence
The relative order of collating elements as determined by the setting of the LC_COLLATE category in the applied FDCC-set

3.1.3.6
equivalence class
A set of collating elements with the same primary collation weight

NOTE: Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight.

3.2 Notations

The following notations and common conventions for specifications apply to this International Standard:

3.2.1 Notation for defining syntax

In this International Standard, the description of an individual record in a FDCC-set is done using the syntax notation given in the following.

The syntax notation looks as follows:

"<format>",[<arg1>,<arg2>,...,<argn>]

The <format> is given in a format string enclosed in double quotes, followed by a number of parameters, separated by commas. It is similar to the format specification defined in the ISO/IEC 9945 standard and the format specification used in C language printf() function. The format of each parameter is given by an escape sequence as follows:

%s	specifies a string
%d	specifies a decimal integer
%c	specifies a character
%o	specifies an octal integer
%x	specifies a hexadecimal integer

A " " (an empty character position) in the syntax string represents one or more <blank> characters.

All other characters in the format string represent themselves, except:

%%	specifies a single %
\n	specifies an end-of-line

The notation "..." is used to specify that repetition of the previous specification is optional, and this is done in both the format string and in the parameter list.

3.2.3 Portable character set

A set of symbolic names for characters in Table 1, which is called the portable character set, is used in character description text of this specification. The first eight entries in Table 1 are defined in ISO/IEC 6429 and the rest is defined in ISO/IEC 9945 with some definitions from ISO/IEC 10646.

Table 1: Portable character set

Symbolic name	Glyph	UCS	Description

<NUL>		<U0000>	NULL (NUL)
<alert>		<U0007>	BELL (BEL)
<backspace>		<U0008>	BACKSPACE (BS)
<tab>		<U0009>	CHARACTER TABULATION (HT)
<carriage-return>		<U000D>	CARRIAGE RETURN (CR)
<newline>		<U000A>	LINE FEED (LF)
<vertical-tab>		<U000B>	LINE TABULATION (VT)
<form-feed>		<U000C>	FORM FEED (FF)
<space>		<U0020>	SPACE
<exclamation-mark>	!	<U0021>	EXCLAMATION MARK
<quotation-mark>	"	<U0022>	QUOTATION MARK
<number-sign>	#	<U0023>	NUMBER SIGN
<dollar-sign>	$	<U0024>	DOLLAR SIGN
<percent-sign>	%	<U0025>	PERCENT SIGN
<ampersand>	&	<U0026>	AMPERSAND
<apostrophe>	'	<U0027>	APOSTROPHE
<left-parenthesis>	(	<U0028>	LEFT PARENTHESIS
<right-parenthesis>	)	<U0029>	RIGHT PARENTHESIS
<asterisk>	*	<U002A>	ASTERISK
<plus-sign>	+	<U002B>	PLUS SIGN
<comma>	,	<U002C>	COMMA
<hyphen-minus>	-	<U002D>	HYPHEN-MINUS
<hyphen>	-	<U002D>	HYPHEN-MINUS
<full-stop>	.	<U002E>	FULL STOP
<period>	.	<U002E>	FULL STOP
<slash>	/	<U002F>	SOLIDUS
<solidus>	/	<U002F>	SOLIDUS
<zero>	0	<U0030>	DIGIT ZERO
<one>	1	<U0031>	DIGIT ONE
<two>	2	<U0032>	DIGIT TWO
<three>	3	<U0033>	DIGIT THREE
<four>	4	<U0034>	DIGIT FOUR
<five>	5	<U0035>	DIGIT FIVE
<six>	6	<U0036>	DIGIT SIX
<seven>	7	<U0037>	DIGIT SEVEN
<eight>	8	<U0038>	DIGIT EIGHT
<nine>	9	<U0039>	DIGIT NINE
<colon>	:	<U003A>	COLON
<semicolon>	;	<U003B>	SEMICOLON
<less-than-sign>	<	<U003C>	LESS-THAN SIGN
<equals-sign>	=	<U003D>	EQUALS SIGN
<greater-than-sign>	>	<U003E>	GREATER-THAN SIGN
<question-mark>	?	<U003F>	QUESTION MARK
<commercial-at>	@	<U0040>	COMMERCIAL AT
<A>	A	<U0041>	LATIN CAPITAL LETTER A
<B>	B	<U0042>	LATIN CAPITAL LETTER B
<C>	C	<U0043>	LATIN CAPITAL LETTER C
<D>	D	<U0044>	LATIN CAPITAL LETTER D
<E>	E	<U0045>	LATIN CAPITAL LETTER E
<F>	F	<U0046>	LATIN CAPITAL LETTER F
<G>	G	<U0047>	LATIN CAPITAL LETTER G
<H>	H	<U0048>	LATIN CAPITAL LETTER H
<I>	I	<U0049>	LATIN CAPITAL LETTER I
<J>	J	<U004A>	LATIN CAPITAL LETTER J
<K>	K	<U004B>	LATIN CAPITAL LETTER K
<L>	L	<U004C>	LATIN CAPITAL LETTER L
<M>	M	<U004D>	LATIN CAPITAL LETTER M
<N>	N	<U004E>	LATIN CAPITAL LETTER N
<O>	O	<U004F>	LATIN CAPITAL LETTER O

<P>	P	<U0050>	LATIN CAPITAL LETTER P
<Q>	Q	<U0051>	LATIN CAPITAL LETTER Q
<R>	R	<U0052>	LATIN CAPITAL LETTER R
<S>	S	<U0053>	LATIN CAPITAL LETTER S
<T>	T	<U0054>	LATIN CAPITAL LETTER T
<U>	U	<U0055>	LATIN CAPITAL LETTER U
<V>	V	<U0056>	LATIN CAPITAL LETTER V
<W>	W	<U0057>	LATIN CAPITAL LETTER W
<X>	X	<U0058>	LATIN CAPITAL LETTER X
<Y>	Y	<U0059>	LATIN CAPITAL LETTER Y
<Z>	Z	<U005A>	LATIN CAPITAL LETTER Z
<left-square-bracket>	[	<U005B>	LEFT SQUARE BRACKET
<backslash>	\	<U005C>	REVERSE SOLIDUS
<reverse-solidus>	\	<U005C>	REVERSE SOLIDUS
<right-square-bracket>	]	<U005D>	RIGHT SQUARE BRACKET
<circumflex-accent>	^	<U005E>	CIRCUMFLEX ACCENT
<circumflex>	^	<U005E>	CIRCUMFLEX ACCENT
<low-line>	_	<U005F>	LOW LINE
<underscore>	_	<U005F>	LOW LINE
<grave-accent>	`	<U0060>	GRAVE ACCENT
<a>	a	<U0061>	LATIN SMALL LETTER A
<b>	b	<U0062>	LATIN SMALL LETTER B
<c>	c	<U0063>	LATIN SMALL LETTER C
<d>	d	<U0064>	LATIN SMALL LETTER D
<e>	e	<U0065>	LATIN SMALL LETTER E
<f>	f	<U0066>	LATIN SMALL LETTER F
<g>	g	<U0067>	LATIN SMALL LETTER G
<h>	h	<U0068>	LATIN SMALL LETTER H
<i>	I	<U0069>	LATIN SMALL LETTER I
<j>	j	<U006A>	LATIN SMALL LETTER J
<k>	k	<U006B>	LATIN SMALL LETTER K
<l>	l	<U006C>	LATIN SMALL LETTER L
<m>	m	<U006D>	LATIN SMALL LETTER M
<n>	n	<U006E>	LATIN SMALL LETTER N
<o>	o	<U006F>	LATIN SMALL LETTER O
<p>	p	<U0070>	LATIN SMALL LETTER P
<q>	q	<U0071>	LATIN SMALL LETTER Q
<r>	r	<U0072>	LATIN SMALL LETTER R
<s>	s	<U0073>	LATIN SMALL LETTER S
<t>	t	<U0074>	LATIN SMALL LETTER T
<u>	u	<U0075>	LATIN SMALL LETTER U
<v>	v	<U0076>	LATIN SMALL LETTER V
<w>	w	<U0077>	LATIN SMALL LETTER W
<x>	x	<U0078>	LATIN SMALL LETTER X
<y>	y	<U0079>	LATIN SMALL LETTER Y
<z>	z	<U007A>	LATIN SMALL LETTER Z
<left-brace>	{	<U007B>	LEFT CURLY BRACKET
<left-curly-bracket>	{	<U007B>	LEFT CURLY BRACKET
<vertical-line>	|	<U007C>	VERTICAL LINE
<right-brace>	}	<U007D>	RIGHT CURLY BRACKET
<right-curly-bracket>	}	<U007D>	RIGHT CURLY BRACKET
<tilde>	~	<U007E>	TILDE

This International Standard may use other symbolic character names than the above in examples, to illustrate the use of the range of symbols allowed by the syntax specified in 4.1.1.

4. FUNCTIONALITY

Functionality to access the information described in this International Standard is described in ISO/IEC 9899 – Programming Language C. In addition the following functions are specified:

4.1 The “strpcoll” function

Synopsis

```
#include <string.h>
int strpcoll(const char *s1, const char *s2, int p);
```

Description

The strpcoll function compares the string pointed to by s1 to the string pointed to by s2, both interpreted as appropriate to the LC_COLLATE category of the current FDCC-set, and to the precision of p.

Returns

The strpcoll function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2, given the precision p, when both are interpreted as appropriate to the current FDCC-set. The precision p is the level of the collation data that will be used, on alphabetic characters, precision=1 will normally regard all versions, including upper case, lowercase and accented versions of a letter as equal; and precision=2 will normally regard upper case and lower case versions of an accented letter as equal, both in composed and decomposed forms; precision=3 would normally distinction between composed and decomposed forms of a letter.

4.2 The “setmedia” function

Synopsis

```
int setmedia(int io, int media, int allow);
```

Description

The setmedia function sets the message interfaces to be used. io gives input or output; input=0, output=1. media gives the media type: text=1, voice=2, gestures=3, selection=4. allow specifies if the combination of io and media will be allowed or not, allow=1, and deny=0.

Returns

The setmedia function returns a 0 if the function executes without error, and a

positive non-zero error code if the function executes with error.

4.3 String, encoding, repertoire, and locale data types

As basic string handling is dependent on the user's preferences (as given via the string), encoding, repertoire, and locale data types are described together here.

4.3.1 String data type

The string handling APIs defined in this document operate on an internal representation of character strings, which are arrays of characters, A void string is indicated by the implementation-defined value NIL. The string data type is defined by:

```
#define string *wchar_t
```

4.3.2 Encoding data type

The "encoding" data type holds data necessary to convert to and from an external encoding and the internal string representation. This includes mapping of coded characters to the internal repertoire, how to shift between subencodings such as via ISO 2022 techniques, or representation via symbolic character names identified via introducing sequences, and state information. The enocding data type is defined by:

```
struct encoding {
        string encodingname;
        // other things ???
}
```

NOTE The encoding definition is closely related to the "charmap" specification of POSIX and TR 14652, the "charset" definition in the Internet MIME specification, and newer developments for the C and C++ programming languages.

4.3.2.1 int newencoding(const string encodingname,encoding enc)

The "newencoding" API creates an encoding object with the necessary space to hold all information necessary to convert between the encoding and the internal string representation. The "newencoding" API sets default values, including the "line_terminator" to being the characters "CR""LF", the "invalid_char" to being the character "SUB", the "symbolic_char_introducer" to being "NUL" (not valid), the "sub_encoding_change" API, the "get_symbolic_char_name" API and the "put_symbolic_char_name" API to be the null API, and the "input_state" and the "output_state" variables to be the initial state .

The "encodingname" is an implementation defined string with the following characterisics:

An initial string of "std/" refers to the charmaps registered in the international cultural register, ISO/IEC 15897.

If the specified encoding is syntactically valid and supported by the implementation, the "newencoding" API allocates memory for the new object and returns a pointer to the object in the parameter "enc". It is the application's responsibility to free this memory with a call to the "freeencoding" API when the struct is no longer needed. If the API fails for any reason, the contents of "enc" is undefined.

The "newencoding" API returns one of the following values:

0 -LC_SUCCESS - The API call was successful

1 - LC_NOTSUPPORTED - The encoding is not supported by the current system.
2 - LC_NOMEMORY - there was insufficient memory to perform the API
3 -LC_INVALID - The specified encoding is invalid

4.3.2.2 int freeencoding(encoding enc)

The "freeencoding" API frees the memory occupied by the encoding "enc". It returns a zero if the operation is succesful, and a -1 otherwise.

4.3.2.4 int setencint(encoding enc,const string param,int val)

iThe "setencint" API sets a specific parameter as specified in the string "param" of the encoding specification to a specific integer value as specified in "val". The defined values for "param" are:
(to be described)
It returns a zero if the operation is succesful, and a 1 otherwise.

4.3.2.4 int setencbytes(encoding enc,const string param,const char *val,int len)

The "setencbytes" API sets a specific parameter as specified in the string "param" of the encoding specification to a specific multibyte value as specified in "val" with the length "len" bytes. The defined values for "param" are:
"line_terminator"
"invalid_char"
"symbolic_char_introducer"

4.3.2.5 int setencproc(encoding enc, const string param, int val())

The "setencproc" API sets a specific parameter as specified in the string "param" of the encoding specification to a specific API value as specified in "val" . The defined values for "param" are:
"sub_encoding_change" - procedure subec()
"get_symbolic_char_name" - procedure gscn(c, p , len)
"put_symbolic_char_name" - procedure pscn(c, p, len)

4.3.2.6 int gscn(c, p, len)

The application defined "get_symbolic_char_name" API is called by the "bytes2string" API when the character sequence in "symbolic_char_introducer" is met in the input octet sequence. It gets a pointer "p" to the first octet after the "symbolic_char_introducer" and determines whether there is a symbolic character according to the application APIs definitions, with or without a terminator sequence, within "len" octets after the "p" pointer. If successful the API returns the found character in the internal string representation in "c" and the pointer "p" to the first octet after the symbolic character, including the possible terminator sequence. The application defined API returns:
0 if succesful
1 if it could not recognise a symbolic character within the "len" octets. "p" is not changed.
2 if the octet sequence is invalid according to the rules of the application. "p" is not changed.

4.3.2.7 int pscn(c, p, len)

The application defined "put_symbolic_char_name" API is called by the "string2bytes" API when a character is not present in the external encoding. It gets a pointer "p" to the next octet to be written in the sequence of octets and determines whether there is room to put a symbolic character

according to the application APIs definitions, with the "symbolic_char_introducer" value and with or without a terminator sequence, within "len" octets after the "p" pointer. If successful, the API returns "p", a pointer, to the first octet after the symbolic character written, including the possible terminator sequence. The application defined API returns:

0 if successful

1 if the API was not able to write the symbolic character within "len" octets. The pointer "p" is not changed.

2 if the API had no means of writing the character "c", The pointer "p" is not changed.

4.3.3 Repertoire data type

The "repertoire" data type holds data necessary for the "stringtrans" transliteration API.

4.3.3.1 int newrepertoire(const string repertoirename,repertoire rep)

The "newrepertoire" API creates a repertoire object with the necessary space to hold all information necessary. The "repertoirename" is an implementation defined string with the following characteristics: An initial string of "std/" refers to repertoiremaps registered in the international cultural register, ISO/IEC 15897. If the specified repertoire is valid and supported, the "newrepertoire" API allocates memory for the new object and returns a pointer to the object in "rep". It is the application's responsibility to free this memory with a call to the "freerepertoire" API when the object is no longer needed. If the API fails for any reason, the contents of "rep" is undefined.

The "newrepertoire" API returns one of the following values:

0 - The API call was successful

1 - The repertoire is not supported by the current system.

2 - There was insufficient memory to perform the API

3 - The specified repertoire is invalid

4.3.3.2 int freerepertoire(repertoire rep)

The "freerepertoire" API frees the memory occupied by the repertoire "rep". It returns 0 if the operation is successful, and 1 otherwise.

4.3.3.3 int enc2repertoire(encoding enc,repertoire rep)

The "enc2repertoire" API generates a repertoire object with a repertoire corresponding to the character repertoire of the encoding "enc". If the API is successful, it returns the repertoire object in "rep". It has the same return values as the "newrepertoire" API.

4.3.4 Locale data type

The "locale" data type is a pointer to a struct with a number of variables capable of holding information sufficient to service all language-dependent internationalization services. The "locale" data type has provisions to affect groups of functionalities in categories, which are:

1 NULL
2 LC_ALL
3 LC_IDENTIFICATION
4 LC_COLLATE
5 LC_CTYPE
6 LC_MONETARY
7 LC_NUMERIC
8 LC_TIME

9 LC_MESSAGES
10 LC_XLITERATE
11 LC_NAME
12 LC_ADDRESS
13 LC_TELEPHONE

The category LC_ALL denotes all of the other non-void categories.
The category NULL denotes a void category.

The "locale" data type includes the following variables (which are further described in this document:)
LC_MONETARY values:
int_curr_symb: string.
currency_symbol: string.
mon_deccimal_point: string.
mon_thousands_sep: string.
mon_grouping: string
positive_sign: string.
negative_sign: string.
int_frac_digits: integer.
frac_digits: integer.
p_cs_precedes: integer
p_sep_by_space: integer.
n_cs_precedes: integer
n_sep_by_space: integer
p_sign_posn: integer
n_sign_posn: integer

LC_NUMERIC values:
decimal_point: string
thousands_sep: string
grouping: array of integers

LC_TIME values:
abday: array (1,7) of string
day: array (1,7) of string
abmon: array (1,13) of string
mon: array (1,13) of string
d_t_fmt: string
d_fmt: string
t_fmt: string
am_pm: string
t_fmt_ampm: string
era: string
era_year: string
era_d_fmt: string
alt_digits: array (1,100) of string
LC_MESSAGES values:

yesexpr: string
noexpr: string

4.3.4.1 int newlocale(int category_mask, const string localename,locale lc)

The "newlocale" API creates a locale struct with all the necessary information to perform the language-sensitive operations of internationalization APIs accepting an argument of the type "locale". If the API is successful, all categories of the locale object are created and initialized. Any categories in the locale identified by "localename" are initalized to the i18n locale.

The "localename" is an implementation-defined value with the following characteristics:

- An initial string of "std/" refers to the locales registered in the international cultural register, ISO/IEC 15897.

If the specified locale is valid and supported, the "newlocale" API allocates memory for the new object and returns a pointer to the object in "lc". It is the application's responsibility to free this memory with a call to the "freelocale" API when the object is no longer needed. If the API fails for any reason, the contents of "lc" is undefined.

The "newlocale" API returns one of the following values:

0 - LC_SUCCESS - The API call was successful.

1 - LC_INCOMPLETE - The specified locale has been created, but the locale object contains one or more categories that were initialized to the i18n locale because the "localename" did not identify a value for that category.

2 - LC_NOTSUPPORTED - The locale is not supported by the current system.

3 - LC_NOMEMORY - there was insufficient memory to perform the API.

4 - LC_INVALID - The specified locale is invalid.

4.3.4.2 int freelocale(locale lc)

The "freelocale" API frees the memory occupied by the locale "lc". It returns 0 if the operation is succesful, and 1 otherwise.

4.3.4.3 int modifylocale(const int category,const string localename,locale lc)

The "modifylocale" API modifies the values of the locale object "lc" parameter relating to the category "category" and with values as specified in "localename". "category" takes values as defined in 5.4 and "localename" is defined as for the "newlocale" API. The return value is as for the "newlocale" API.

4.3.4.4 int intlocaleinfo(const int category,const string keywordname,locale lc)

The "intlocaleinfo" API gets the integer value of the keyword "keywordname" of the locale object "lc" relating to the category "category". "category" takes values as defined in 5.4.The return value is the integer value of the keyword.

4.3.4.5 string stringlocaleinfo(const int category,const string keywordname,locale lc)

The "stringlocaleinfo" API gets the string value of the keyword "keywordname" of the locale object "lc" relating to the category "category". "category" takes values as defined in 5.4.The return value is the string value of the keyword.

4.3.5 Character handling

The character handling APIs behave according to the LC_CTYPE category of the locale parameter for the individual APIs.

4.3.5.1 int istype(wchar_t c,const string c_type,const locale lc)

The "istype" API returns 1 if the character "c" is in the type "c_type", else 0.
"c_type" can have the following values:
alnum, alpha, cntrl, digit, graph, lower, print, punct, space, blank, upper, xdigit

4.3.5.2 int tolowers(string s1,const string s2,const locale lc)

The "tolowers" API returns in string "s1" all characters in the string "s2" converted to the corresponding lowercase characters with conversion rules given by the locale "lc". The API returns the number of resulting characters in "s1".

4.3.5.3 int touppers(string s1,const string s2,const locale lc)

The "touppers" API returns in string "s1" all characters in the string "s2" converted to the corresponding uppercase characters with conversion rules given by the locale "lc". The API returns the number of resulting charactes in "s1".

4.3.5.4 int stringtrans(transtype, maxlen,string s1,const string s2, rep)

The "stringtrans" API transforms string "s2" into string "s1" given the transformation specifications as noted below.
Values for the "transtype" parameter are
1 - as for the "tolowers" API
2 - as for the "touppers" API
3 - transliterate the string "s2" into the string "s1" (for example using for each character the first "transform" specification of ISO/IEC TR 14652) that is using the repertoire of "rep" and has at most "maxlen" characters as the transliteration. If the "s1" string is to be exceeded, or there is no valid transliteration, the API returns -1. Otherwise it returns the resulting number of characters of "s1".

4.3.6 String comparison

The string comparison APIs behave according to the LC_COLLATE category of the locale parameter for the individual APIs.

4.3.6.1 int strcoll_l(const string s1,const string s2,locale lc)

int strncoll_l(const string s1,const string s2, n,locale lc)

The "strcoll_l" API compares the two strings "s1" and "s2" with regards to the collating specifications of the locale "lc".
The "strncoll_l" API compares at most "n" characters of the two strings "s1" and "s2" with regards to the collating specifications of the locale "lc".
Both the "strcoll_l" and "strncoll_l" APIs returns -1 if "s1" < "s2", 0 if "s1" == "s2" and 1 if "s1" > "s2" .

4.3.6.2 int strxfrm_l(const string s1,const string s2,locale lc)

The "stringxfrm" API converts the character string "s2" using the locale "lc" and to the precision in "precision"as defined in 4.2, to an internal representation in "s1" suitable for comparison via a binary comparison API (in C this may be strcmp()).

4.3.6.3 int stringcoll(const string s1,const string s2,int precision,locale lc)

4.3.6.4 int stringncoll(const string s1,const string s2,int precision,int n,locale lc)

The "stringcoll" API compares the two strings "s1" and "s2" with regards to the collating specifications of the locale "lc" and to the precision in "precision".

The "stringncoll" API compares at most "n" characters of the two strings "s1" and "s2" with regards to the collating specifications of the locale "lc" and to the precision in "precision".

The "precision" indicates to what level of preciseness the string comparison is done. "precision" may have the following values:

0 - all levels
1 - only to level 1 - CASE_AND_ACCENT_INSENSITITVE
2 - only to level 2 - CASE_INSENSITIVE
3 - only to level 3 - IGNORE_SPECIALS
4 - only to level 4 - EXACT_MATCHING

Both the "stringcoll" and "stringncoll" APIs returns -1 if "s1" < "s2", 0 if "s1" == "s2" and 1 if "s1" > "s2" .

4.3.6.5 int stringxfrm(const string s1,const string s2,int precision,locale lc)

The "stringxfrm" API converts the character string "s2" using the locale "lc" and to the precision in "precision" as defined in 4.2, to an internal representation in "s1" suitable for comparison via a binary comparison API (in C this may be strcmp()).

4.3.7 Message formatting

The message formatting APIs behave according to the LC_MESSAGES category of the locale parameter for the individual APIs.

4.3.7.1 string stringget(const string msgtag,const string textdomain,locale lc)

The "stringget" API gets the message with the tag "msgtag" in the current LC-MESSAGES part of the "lc" locale with respect to the "textdomain" set of messages. If not found or the locale is invalid and no "msgtag" is found in the default locale, then "msgtag" is returned.

4.3.8 Conversion between string and other data types

4.3.8.1 int string2int_l(string s,locale lc)

The "string2int_l" API converts a string to an integer, with respect to the locale "lc".

4.3.8.2 string int2string_l(int i,locale lc)

The "int2string_l" API creates a string with the necessary length and returns the string with an integer formatted in characters, according to the locale "lc". If there is not enough memory to create a new string, the API returns the void string.

4.3.8.3 double string2real_l(string s,locale lc)

The "string2real_l" API converts a string to a real value, using information about thousands and decimal separators from the locale "lc". If there is not enough memory to create a new string, the API returns the void string.

4.3.8.4 string real2string_l(double r,locale lc)

The "real2string_l" API formats a real value into a string, with decimal and thousands separators as given in the "lc" locale. Returns a string with the necessary length, or if memory is not available it returns the empty string.

4.3.8.5 int bytes2string_e(string s,char* p,int len,encoding enc)

The "bytes2string_e" API converts "len" octets from the multibyte value "p" in the encoding "enc" to the string "s", and with the conversion input_state as recorded in "enc". The conversion stops earlier in two cases: if the next character to be stored in the string "s" would exceed the length of "s", or if there is a sequence not corresponding to a recognizable character in the input sequence of octets, possibly after calling an application-defined "get_symbolic_char" API.

If the API stops without having converted "len" octets, the API returns the negative to the number of octets converted. Otherwise it returns the number of internal characters converted (ie. the last index in the string "s" for characters converted).

4.3.8.6 int string2bytes_e(char* p,string s,int len,encoding enc)

The "string2bytes_e" API converts a string "s" into a sequence of corresponding octets of "p" in the encoding "enc", and beginning in the output_state recorded in "enc". The conversion continues up to the lenght of the string "s". The conversion stops earlier in two cases: when a code is reached that does not correspond to a valid representation in the sequence of octets, and either no "invalid_char" value or "put_symbolic_char" API is defined, or the application defined "put_symbolic_char" API returns with a value 2; or the next octet would exceed the limit of "len" total octets to be stored in the multibyte "p" variable.

The API returns the negative index of the character in question if the conversion stops because it could not convert a character in the string to octets. Otherwise it returns the number of octets in the resulting sequence of octets.

4.3.8.7 int time2string_l(string s,const string format,const struct tm *timeptr,locale lc)

The "time2string_l" API returns a string "s" formatted according to the format in "format" of the time value in "timeptr", according to the local conventions in the locale "lc". The "format" string is specified in IS 9945 (with extensions as described in this document) as the "d_t_fmt" specification.

4.3.8.8 int string2time_l(const struct tm *time,string s,locale lc)

The "string2time_l" API returns a binary time in "time", scanned from the string "s" according to the locale conventions in the locale "lc".

NOTE: This specification needs more work. The C++ standard is the only standard having provisions for this, but is very weak on the subject.

4.3.8.9 int money2string_l(string s,const string format,const double amount,const struct tm *timeptr ,locale lc)

API "money2string_l" returns in parameter "s" a string formatted according to the format in "format" of the money value "amount". The formatting is done with respect to the locale "lc" at the time given in "time". The return value is the number of characters formatted, or -1 if an error occurred.

The parameter "format" is a string that consist of characters that shall be transfered to the output string "s" literally, and formatting specifications that specifies how the money value "amount" is to be formatted.

A formatting specification consist of the following sequence:

- a "%" character
- optional flags
- optional field width
- optional left precision
- optional right position
- the formatting character to specify which formatting to perform.

Flags are given with special characters, width and precision information is given with decimal digits, and formating characters are given with latin letters or the character "%".

The flags are:

"=f" an "=" followed by a single character which is used as the numeric fill character. No restriction is made on the representation of this single character. The default numeric fill character is the SPACE character. This flag does not affect the field width filling which always uses the SPACE character. This flag is ignored unless a left precision (se below) is specified.

"^" Do not use the grouping characters when formatting the amount. The default is to insert the grouping characters if defined in the locale "lc".

"+" or "(" Specify the style of representing positive and negative amounts. Only one of "+" or "(" may be specified. If "+" is specified, the equivalent of "+" and "-" are used from the locale "lc". If "(" is specified, negative amounts are enclosed within parenthesis. If neither flag is specified, the "+" style is used.

"!" Suppersses the currency symbol from the output conversion.

"-" Specify the alignment. If this flag is present all fields are left-justified (padded to the right) rather than right-justified.

Field Width

w A string of decimal digits specifying the minimum field width in characters in which the result of the conversion is right-justified (or left-justified if the "-" flag is specified) The default is 0.

Left Precision

"#n" A "#" followed by a string of decimal digits specifying a maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from several calls to API "money2string" aligned in the same coloumns. It can also be used to fill unused positions with a special character specified with the "=f" flag, as in $***123.45. If more than "n" positions are required, this formatting specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character, see the "=f" flag above.

If grouping has not been suppressed with the "^" flag, and it is defined for the locale "lc", grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters, even if the fill character is a digit.

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign, symbols are padded as neccessary with SPACE characters to make their

positive or negative formats an equal length.

Right precision

".p" A "." followed by a string of decimal digits specifying the number after the radix character. If the value of the right precision is 0, no radix character appears. If a right precision is not included, the value specified in the "lc" locale is used. It is recommended to normally use the value from the locale. The amount being formatted is rounded to the specified number of digits prior to formatting.

Formatting characters:

The formatting characters and their meanings are:

"d" The following characters and up to any corresponding "%d" or the end of the formatting string are only interpreted if there is a second currency in the "lc" locale for the time "t". The amount "a" is converted according to the "conversion_rate" of the locale "lc" and formatted according to any following formatting characters. No argument is converted. There shall be no flags, nor width or precision parameters, just "%%" is allowed.

"i" The type money argument is formatted acording to the "lc" locales's international currency format.

"n" The type money argument is formatted acording to the "lc" locales's national currency format.

"%" Convert to a "%"; no argument is converted. There shall be no flags, nor width or precision parameters, just "%%" is allowed.

4.3.8.10 int name2string_l(string s,const string format,const string name,locale lc)

The API “name2string_l” formats a set of personal name information as given in "name" to a string "s" according to the format in"format" and to the locale given in "lc". The format specification is unspecified (but a description may be found in TR 14652 for the keyword "name_fmt") if this is the empty string, the format specified in the "name_fmt" keyword of the locale in "lc" is used. The return value is the number of characters in the resulting string "s" or -1 if the supplied string "s" had indufficient length to hold the result.

The namerecord shall contain the following strings, which may each be empty:

family - family names, corresponding to the %f and %F escape sequence
given - first given name
giveninit - initial of given name
middle - middle names
middleinit - middle initials
shortname - a shorter name, eg. "Bill"
profession - the profession title
salutation - common salutation, like "Mr."
intsalut - a string with a digit in the range 1 to 5 for salutation

Similar strings with a "r" prepended to the name shall be present to hold Romanized information on the above items.

4.3.8.11 int address2string_l(string s,const string format,const string address,locale lc)

API “address2string_l” formats a set of address information as given in "address" to a string "s" according to the format in"format" and to the locale given in "lc". The format specification is unspecified, (but a description may be found in TR 14652 for the keyword "postal_fmt"); if this is the empty string, the format specified in the "postal_fmt" keyword of the locale in "lc" is used. The return value is the number of characters in the resulting string "s" - or -1 if an error occurred.

The addressrecord shall contain the following strings (with corresponding escape sequences

given in parentheses), which may each be empty:
co - C/o address (%a)
firm - firm name (%f)
department - department name (%d)
building - building name (%b)
streetblock - street or block name (%s)
house - house number or designation (%h)
room - room number or designation (%r)
floor - floor number (%e)
village - village name (%v)
town - town or city name (%T)
countrycode - country designation or code (%C)
zip - zip or postal code (%z)
country- country name (%c)
Similar strings with a "r" prepended to the name shall be present to hold Romanized information on the above items.

4.3.8.12 int teldom2string_l(string s,const string format,const string telephone,locale lc)

API teldom2string_l formats for domestic use a telephone number as given in "telephone" to a string "s" according to the format in"format" and to the locale given in "lc". The format specification is unspecified (but a description may be found in TR 14652 for the keyword "tel_dom_fmt"); if this is the empty string, the format specified in the "tel_dom_fmt" keyword of the locale in "lc" is used. The return value is the number of characters in the resulting string "s" - or -1 if an error occurred.

4.3.8.13 int telint2string_l(string s,const string format,const string telephone,locale lc)

The API telint2string_l formats for international use a telephone number as given in "telephone" to a string "s" according to the format in"format" and to the locale given in "lc". The format specification is unspecified (but a description may be found in TR 14652 for the keyword "tel_int_fmt"); if this is the empty string, the format specified in the "tel_int_fmt" keyword of the locale in "lc" is used. The return value is the number of characters in the resulting string "s" - or -1 if an error occurred.

Bibliography

ISO/IEC TR 30112: 2014 Information Technology – Specification methods for cultural conventions.