NAME

locale – subset of a user's environment that depends on language and cultural conventions

DESCRIPTION

A *locale* is the definition of the subset of a user's environment that depends on language and cultural conventions. It is made up from one or more categories. Each category is identified by its name and controls specific aspects of the behavior of components of the system. Category names correspond to the following environment variable names:

LC_CTYPE	Character classification and case conversion.
LC_COLLATE	Collation order.
LC_TIME	Date and time formats.
LC_NUMERIC	Numeric formatting.
LC_MONETARY	Monetary formatting.
LC_MESSAGES	Formats of informative and diagnostic messages and interactive responses.

The standard utilities base their behavior on the current locale, as defined in the **ENVIRONMENT** section for each utility. The behavior of some of the C-language functions will also be modified based on the current locale, as defined by the last call to **setlocale**(3C).

Locales other than those supplied by the implementation can be created by the application via the **localedef**(1) utility. The value that is used to specify a locale when using environment variables will be the string specified as the *name* operand to **localedef** when the locale was created. The strings "C" and "POSIX" are reserved as identifiers for the POSIX locale.

Applications can select the desired locale by invoking the **setlocale( )** function with the appropriate value. If the function is invoked with an empty string, such as:

```
setlocale(LC_ALL, "");
```

the value of the corresponding environment variable is used. If the environment variable is unset or is set to the empty string, the **setlocale( )** function sets the appropriate environment.

Locale Definition

Locales can be described with the file format accepted by the **localedef** utility.

The locale definition file must contain one or more locale category source definitions, and must not contain more than one definition for the same locale category.

A category source definition consists of a category header, a category body and a category trailer. A category header consists of the character string naming of the category, beginning with the characters **LC_**. The category trailer consists of the string **END**, followed by one or more blank characters and the string used in the corresponding category header.

The category body consists of one or more lines of text. Each line contains an identifier, optionally followed by one or more operands. Identifiers are either keywords, identifying a particular locale element, or collating elements. Each keyword within a locale must have a unique name (that is, two categories cannot have a commonly-named keyword); no keyword can start with the characters **LC_**. Identifiers must be separated from the operands by one or more blank characters.

Operands must be characters, collating elements or strings of characters. Strings must be enclosed in double-quotes. Literal double-quotes within strings must be preceded by the **<***escape character***>**, described below. When a keyword is followed by more than one operand, the operands must be separated by semicolons; blank characters are allowed both before and after a semicolon.

The first category header in the file can be preceded by a line modifying the comment character. It has the following format, starting in column 1:

"comment_char %c\n",<*comment character***>**

The comment character defaults to the number sign (**#**). Blank lines and lines containing the **<***comment character***>** in the first position are ignored.

The first category header in the file can be preceded by a line modifying the escape character to be used in the file. It has the following format, starting in column 1:

"escape_char %c\n",<*escape character***>**

The escape character defaults to backslash.

A line can be continued by placing an escape character as the last character on the line; this continuation character will be discarded from the input. Although the implementation need not accept any one portion of a continued line with a length exceeding **{LINE_MAX}** bytes, it places no limits on the accumulated length of the continued line. Comment lines cannot be continued on a subsequent line using an escaped newline character.

Individual characters, characters in strings, and collating elements must be represented using symbolic names, as defined below. In addition, characters can be represented using the characters themselves or as octal, hexadecimal or decimal constants. When non-symbolic notation is used, the resultant locale definitions will in many cases not be portable between systems. The left angle bracket (**<**) is a reserved symbol, denoting the start of a symbolic name; when used to represent itself it must be preceded by the escape character. The following rules apply to character representation:

1. A character can be represented via a symbolic name, enclosed within angle brackets **<** and **>**. The symbolic name, including the angle brackets, must exactly match a symbolic name defined in the charmap file specified via the **localedef** −**f** option, and will be replaced by a character value determined from the value associated with the symbolic name in the charmap file. The use of a symbolic name not found in the charmap file constitutes an error, unless the category is **LC_CTYPE** or **LC_COLLATE,** in which case it constitutes a warning condition (see **localedef**(1) for a description of action resulting from errors and warnings). The specification of a symbolic name in a **collating-element** or **collating-symbol** section that duplicates a symbolic name in the charmap file (if present) is an error. Use of the escape character or a right angle bracket within a symbolic name is invalid unless the character is preceded by the escape character.

Example:

<c>;<c–cedilla> "**<M><a><y>**"

2. A character can be represented by the character itself, in which case the value of the character is implementation-dependent. Within a string, the double-quote character, the escape character and the right angle bracket character must be escaped (preceded by the escape character) to be interpreted as the character itself. Outside strings, the characters

   , ; < > *escape_char*

   must be escaped to be interpreted as the character itself.

   Example:

   c β "May"

3. A character can be represented as an octal constant. An octal constant is specified as the escape character followed by two or more octal digits. Each constant represents a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   Example:

   \143;\347;\143\150 "\115\141\171"

4. A character can be represented as a hexadecimal constant. A hexadecimal constant is specified as the escape character followed by an **x** followed by two or more hexadecimal digits. Each constant represents a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   Example:

   \x63;\xe7;\x63\x68 "\x4d\x61\x79"

5. A character can be represented as a decimal constant. A decimal constant is specified as the escape character followed by a **d** followed by two or more decimal digits. Each constant represents a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   Example:

   \d99;\d231;\d99\d104 "\d77\d97\d121"

Only characters existing in the character set for which the locale definition is created can be specified, whether using symbolic names, the characters themselves, or octal, decimal or hexadecimal constants. If a charmap file is present, only characters defined in the charmap can be specified using octal, decimal or hexadecimal constants. Symbolic names not present in the charmap file can be specified and will be ignored, as specified under item 1 above.

LC_CTYPE

The **LC_CTYPE** category defines character classification, case conversion and other character attributes. In addition, a series of characters can be represented by three adjacent periods representing an ellipsis symbol (. . .). The ellipsis specification is interpreted as meaning that all values between the values preceding and following it represent valid characters. The ellipsis specification is valid only within a single encoded character set; that is, within a group of characters of the same size. An ellipsis is interpreted as including in the list all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value of the character following the ellipsis.

Example:

\x30;. . .;\x39;

includes in the character class all characters with encoded values between the endpoints.

The following keywords are recognized. In the descriptions, the term ''automatically included'' means that it is not an error either to include or omit any of the referenced characters.

The character classes **digit**, **xdigit**, **lower**, **upper**, and **space** have a set of automatically included characters. These only need to be specified if the character values (that is, encoding) differ from the implementation default values.

LC_CTYPE1
Begin definition of supplementary codeset 1.

LC_CTYPE2
Begin definition of supplementary codeset 2.

LC_CTYPE3
Begin definition of supplementary codeset 3.

cswidth Report byte count and screen width information.

upper Define characters to be classified as upper-case letters.

In the POSIX locale, the 26 upper-case letters are included:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct**, or **space** can be specified. The upper-case letters **A** to **Z** are automatically included in this class.

lower Define characters to be classified as lower-case letters.

In the POSIX locale, the 26 lower-case letters are included:

a b c d e f g h i j k l m n o p q r s t u v w x y z

In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct**, or **space** can be specified. The lower-case letters **a** to **z** of the portable character set are automatically included in this class.

alpha Define characters to be classified as letters.

In the POSIX locale, all characters in the classes **upper** and **lower** are included. In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct**, or **space** can be specified. Characters classified as either **upper** or **lower** are automatically included in this class.

digit Define the characters to be classified as numeric digits.

In the POSIX locale, only

0 1 2 3 4 5 6 7 8 9

are included.

In a locale definition file, only the digits **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, and **9** can be specified, and in contiguous ascending sequence by numerical value. The digits **0** to **9** of the portable character set are automatically included in this class.

The definition of character class **digit** requires that only ten characters; the ones defining digits can be specified; alternative digits (for example, Hindi or Kanji) cannot be specified here.

space Define characters to be classified as white-space characters.

In the POSIX locale, at a minimum, the characters SPACE, FORMFEED, NEWLINE, CARRIAGE RETURN, TAB, and VERTICAL TAB are included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **graph**, or **xdigit** can be specified. The characters SPACE, FORMFEED, NEWLINE, CARRIAGE RETURN, TAB, and VERTICAL TAB of the portable character set, and any characters included in the class **blank** are automatically included in this class.

cntrl Define characters to be classified as control characters.

In the POSIX locale, no characters in classes **alpha** or **print** are included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **punct**, **graph**, **print**, or **xdigit** can be specified.

punct Define characters to be classified as punctuation characters.

In the POSIX locale, neither the space character nor any characters in classes **alpha**, **digit**, or **cntrl** are included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **cntrl**, **xdigit** or as the space character can be specified.

graph Define characters to be classified as printable characters, not including the space character.

In the POSIX locale, all characters in classes **alpha**, **digit**, and **punct** are included; no characters in class **cntrl** are included.

In a locale definition file, characters specified for the keywords **upper**, **lower**, **alpha**, **digit**, **xdigit**, and **punct** are automatically included in this class. No character specified for the keyword **cntrl** can be specified.

print Define characters to be classified as printable characters, including the space character.

In the POSIX locale, all characters in class **graph** are included; no characters in class **cntrl** are included.

In a locale definition file, characters specified for the keywords **upper**, **lower**, **alpha**, **digit**, **xdigit**, **punct**, and the space character are automatically included in this class. No character specified for the keyword **cntrl** can be specified.

xdigit Define the characters to be classified as hexadecimal digits.

In the POSIX locale, only:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

are included.

In a locale definition file, only the characters defined for the class **digit** can be specified, in contiguous ascending sequence by numerical value, followed by one or more sets of six characters representing the hexadecimal digits 10 to 15 inclusive, with each set in ascending order (for example **A**, **B**, **C**, **D**, **E**, **F**, **a**, **b**, **c**, **d**, **e**, **f**). The digits **0** to **9**, the upper-case letters **A** to **F** and the lower-case letters **a** to **f** of the portable character set are automatically included in this class.

The definition of character class **xdigit** requires that the characters included in character class **digit** be included here also.

blank Define characters to be classified as blank characters.
In the POSIX locale, only the space and tab characters are included.

In a locale definition file, the characters space and tab are automatically included in this class.

toupper Define the mapping of lower-case letters to upper-case letters.

In the POSIX locale, at a minimum, the 26 lower-case characters:

a b c d e f g h i j k l m n o p q r s t u v w x y z

are mapped to the corresponding 26 upper-case characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

In a locale definition file, the operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma and the pair enclosed by parentheses. The first character in each pair is the lower-case letter, the second the corresponding upper-case letter. Only characters specified for the keywords **lower** and **upper** can be specified. The lower-case letters **a** to **z**, and their corresponding upper-case letters **A** to **Z**, of the portable character set are automatically included in this mapping, but only when the **toupper** keyword is omitted from the locale definition.

tolower Define the mapping of upper-case letters to lower-case letters.
In the POSIX locale, at a minimum, the 26 upper-case characters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

are mapped to the corresponding 26 lower-case characters:

a b c d e f g h i j k l m n o p q r s t u v w x y z

In a locale definition file, the operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma and the pair enclosed by parentheses. The first character in each pair is the upper-case letter, the second the corresponding lower-case letter. Only characters specified for the keywords **lower** and **upper** can be specified. If the **tolower** keyword is omitted from the locale definition, the mapping will be the reverse mapping of the one specified for **toupper**.

LC_COLLATE

The **LC_COLLATE** category provides a collation sequence definition for numerous utilities (such as **sort**(1), **uniq**(1), and so forth), regular expression matching (see **regex**(5)), and the **strcoll**(3C), **strxfrm**(3C), **wcscoll**(3I), and **wcsxfrm**(3I) functions.

A collation sequence definition defines the relative order between collating elements (characters and multi-character collating elements) in the locale. This order is expressed in terms of collation values; that is, by assigning each element one or more collation values (also known as collation weights). At least the following capabilities are provided:

1. **Multi-character collating elements**. Specification of multi-character collating elements (that is, sequences of two or more characters to be collated as an entity).
2. **User-defined ordering of collating elements**. Each collating element is assigned a collation value defining its order in the character (or basic) collation sequence. This ordering is used by regular expressions and pattern matching and, unless collation weights are explicity specified, also as the collation weight to be used in sorting.
3. **Multiple weights and equivalence classes**. Collating elements can be assigned one or more (up to the limit **{COLL_WEIGHTS_MAX})** collating weights for use in sorting. The first weight is hereafter referred to as the primary weight.
4. **One-to-Many mapping**. A single character is mapped into a string of collating elements.
5. **Equivalence class definition**. Two or more collating elements have the same collation value (primary weight).
6. **Ordering by weights**. When two strings are compared to determine their relative order, the two strings are first broken up into a series of collating elements; the elements in each successive pair of elements are then compared according to the relative primary weights for the elements. If equal, and more than one weight has been assigned, then the pairs of collating elements are recompared according to the relative subsequent weights, until either a pair of collating elements compare unequal or the weights are exhausted.

The following keywords are recognized in a collation sequence definition. They are described in detail in the following sections.

collating-element Define a collating-element symbol representing a multi-character collating element. This keyword is optional.

collating-symbol Define a collating symbol for use in collation order statements. This keyword is optional.

order_start Define collation rules. This statement is followed by one or more collation order statements, assigning character collation values and collation weights to collating elements.

order_end Specify the end of the collation-order statements.

collating-element *keyword*

In addition to the collating elements in the character set, the **collating-element** keyword is used to define multi-character collating elements. The syntax is:

"collating-element %s from \"%s\"\n",<*collating-symbol*>,<*string*>

The <*collating-symbol*> operand is a symbolic name, enclosed between angle brackets (**<** and **>**), and must not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. The string operand is a string of two or more characters that collates as an entity. A <*collating-element*> defined via this keyword is only recognized with the **LC_COLLATE** category.

Example:

collating-element <ch> from "**<c><h>**"
collating-element <e-acute> from "**<acute><e>**"
collating-element <ll> from "**ll**"

collating-symbol *keyword*

This keyword will be used to define symbols for use in collation sequence statements; that is, between the **order_start** and the **order_end** keywords. The syntax is:

"collating-symbol %s\n",<*collating-symbol*>

The <*collating-symbol*> is a symbolic name, enclosed between angle brackets (**<** and **>**), and must not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition.

A **collating-symbol** defined via this keyword is only recognized with the **LC_COLLATE** category.

Example:

collating-symbol <UPPER_CASE>
collating-symbol <HIGH>

The **collating-symbol** keyword defines a symbolic name that can be associated with a relative position in the character order sequence. While such a symbolic name does not represent any collating element, it can be used as a weight.

order_start *keyword*

The **order_start** keyword must precede collation order entries and also defines the number of weights for this collation sequence definition and other collation rules.

The syntax of the **order_start** keyword is:

"order_start %s;%s;. . .;%s\n",<*sort-rules*>,<*sort-rules*>

The operands to the **order_start** keyword are optional. If present, the operands define rules to be applied when strings are compared. The number of operands define how many weights each element is assigned; if no operands are present, one **forward** operand is assumed. If present, the first operand defines rules to be applied when comparing strings using the first (primary) weight; the second when comparing strings using the second weight, and so on. Operands are separated by semicolons (**;**). Each operand consists of one or more collation directives, separated by commas (**,**). If the number of operands exceeds the **{COLL_WEIGHTS_MAX}** limit, the utility will issue a warning message. The following directives will be supported:

forward Specifies that comparison operations for the weight level proceed from start of string towards the end of string.

backward Specifies that comparison operations for the weight level proceed from end of string towards the beginning of string.

position Specifies that comparison operations for the weight level will consider the relative position of elements in the strings not subject to **IGNORE.** The string containing an element not subject to **IGNORE** after the fewest collating elements subject to **IGNORE** from the start of the compare will collate first. If both strings contain a character not subject to **IGNORE** in the same relative position, the collating values assigned to the elements will determine the ordering. In case of equality, subsequent characters not subject to **IGNORE** are considered in the same manner.

The directives **forward** and **backward** are mutually exclusive.

Example:

order_start forward;backward

If no operands are specified, a single **forward** operand is assumed.

The character (and collating element) order is defined by the order in which characters and elements are specified between the **order_start** and **order_end** keywords. This character order is used in range expressions in regular expressions (see **regex**(5)). Weights assigned to the characters and elements define the collation sequence; in the absence of weights, the character order is also the collation sequence.

The **position** keyword provides the capability to consider, in a compare, the relative position of characters not subject to **IGNORE.** As an example, consider the two strings ‘‘o-ring’’ and ‘‘or-ing’’. Assuming the hyphen is subject to **IGNORE** on the first pass, the two strings will compare equal, and the position of the hyphen is immaterial. On second pass, all characters except the hyphen are subject to **IGNORE,** and in the normal case the two strings would again compare equal. By taking position into account, the first collates before the second.

Collation Order

The **order_start** keyword is followed by collating identifier entries. The syntax for the collating element entries is

"%s %s;%s;. . .;%s\n"<*collating-identifier*>,<*weight*>,<*weight*>,**. . .**

Each *collating-identifier* consists of either a character described in **Locale Definition** above, a <*collating-element*>, a <*collating-symbol*>, an ellipsis, or the special symbol **UNDEFINED**. The order in which collating elements are specified determines the character order sequence, such that each collating element compares less than the elements following it. The NUL character compares lower than any other character.

A <*collating-element*> is used to specify multi-character collating elements, and indicates that the character sequence specified via the <*collating-element*> is to be collated as a unit and in the relative order specified by its place.

A <*collating-symbol*> is used to define a position in the relative order for use in weights. No weights are specified with a <*collating-symbol*>.

The ellipsis symbol specifies that a sequence of characters will collate according to their encoded character values. It is interpreted as indicating that all characters with a coded character set value higher than the value of the character in the preceding line, and lower than the coded character set value for the character in the following line, in the current coded character set, will be placed in the character collation order between the previous and the following character in ascending order according to their coded character set values. An initial ellipsis is interpreted as if the preceding line specified the NUL character, and a trailing ellipsis as if the following line specified the highest coded character set value in the current coded character set. An ellipsis is treated as invalid if the preceding or following lines do not specify characters in the current coded character set.

The symbol **UNDEFINED** is interpreted as including all coded character set values not specified explicitly or via the ellipsis symbol. Such characters are inserted in the character collation order at the point indicated by the symbol, and in ascending order according to their coded character set values. If no **UNDEFINED** symbol is specified, and the current coded character set contains characters not specified in this section, the utility will issue a warning message and place such characters at the end of the character collation order.

The optional operands for each collation-element are used to define the primary, secondary, or subsequent weights for the collating element. The first operand specifies the relative primary weight, the second the relative secondary weight, and so on. Two or more collation-elements can be assigned the same weight; they belong to the same *equivalence class* if they have the same primary weight. Collation behaves as if, for each weight level, elements subject to **IGNORE** are removed, unless the **position** collation directive is specified for the corresponding level with the **order_start** keyword. Then each successive pair of elements is compared according to the relative weights for the elements. If the two strings compare equal, the process is repeated for the next weight level, up to the limit {**COLL_WEIGHTS_MAX**}.

Weights are expressed as characters described in **Locale Definition** above, <*collating-symbol*>s, <*collating-element*>s, an ellipsis, or the special symbol **IGNORE.** A single character, a <*collating–symbol*> or a <*collating–element*> represent the relative position in the character collating sequence of the character or symbol, rather than the character or characters themselves. Thus, rather than assigning absolute values to weights, a particular weight is expressed using the relative order value assigned to a collating element based on its order in the character collation sequence.

One-to-many mapping is indicated by specifying two or more concatenated characters or symbolic names. For example, if the character **<eszet>** is given the string "**<s><s>**" as a weight, comparisons are performed as if all occurrences of the character **<eszet>** are replaced by **<s><s>** (assuming that **<s>** has the collating weight **<s>**). If it is necessary to define **<eszet>** and **<s><s>** as an equivalence class, then a collating element must be defined for the string **ss**.

All characters specified via an ellipsis will by default be assigned unique weights, equal to the relative order of characters. Characters specified via an explicit or implicit **UNDEFINED** special symbol will by default be assigned the same primary weight (that is, belong to the same equivalence class). An ellipsis symbol as a weight is interpreted to mean that each character in the sequence has unique weights, equal to the relative order of their character in the character collation sequence. The use of the ellipsis as a weight is treated as an error if the collating element is neither an ellipsis nor the special symbol **UNDEFINED.**

The special keyword **IGNORE** as a weight indicates that when strings are compared using the weights at the level where **IGNORE** is specified, the collating element is ignored; that is, as if the string did not contain the collating element. In regular expressions and pattern matching, all characters that are subject to **IGNORE** in their primary weight form an equivalence class.

An empty operand is interpreted as the collating element itself.

For example, the order statement:

```
<a>     <a>;<a>
```

is equal to:

```
<a>
```

An ellipsis can be used as an operand if the collating element was an ellipsis, and is interpreted as the value of each character defined by the ellipsis.

The collation order as defined in this section defines the interpretation of bracket expressions in regular expressions.

Example:

```
order_start     forward;backward
UNDEFINED       IGNORE;IGNORE
<LOW>
<space>         <LOW>;<space>
...             <LOW>;...
<a>             <a>;<a>
<a-acute>       <a>;<a-acute>
<a-grave>       <a>;<a-grave>
<A>             <a>;<A>
```

<A-acute>	**<a>;<A-acute>**
<A-grave>	**<a>;<A-grave>**
<ch>	**<ch>;<ch>**
<Ch>	**<ch>;<Ch>**
<s>	**<s>;<s>**
<eszet>	**"<s><s>";"<eszet><eszet>"**
order_end	

This example is interpreted as follows:

1. The **UNDEFINED** means that all characters not specified in this definition (explicitly or via the ellipsis) are ignored for collation purposes; for regular expression purposes they are ordered first.
2. All characters between **<space>** and **<a>** have the same primary equivalence class and individual secondary weights based on their ordinal encoded values.
3. All characters based on the upper– or lower–case character **a** belong to the same primary equivalence class.
4. The multi-character collating element **<ch>** is represented by the collating symbol **<ch>** and belongs to the same primary equivalence class as the multi-character collating element **<Ch>**.

order_end *keyword*

The collating order entries must be terminated with an **order_end** keyword.

LC_MONETARY

The **LC_MONETARY** category defines the rules and symbols that are used to format monetary numeric information. This information is available through the **localeconv**(3C) function

The following items are defined in this category of the locale. The item names are the keywords recognized by the **localedef**(1) utility when defining a locale. They are also similar to the member names of the **lconv** structure defined in **<locale.h>**. The **localeconv** function returns **{CHAR_MAX}** for unspecified integer items and the empty string (**""**) for unspecified or size zero string items.

In a locale definition file the operands are strings. For some keywords, the strings can contain only integers. Keywords that are not provided, string values set to the empty string (**""**), or integer keywords set to –**1**, are used to indicate that the value is not available in the locale.

int_curr_symbol — The international currency symbol. The operand is a four-character string, with the first three characters containing the alphabetic international currency symbol in accordance with those specified in the ISO 4217:1987 standard. The fourth character is the character used to separate the international currency symbol from the monetary quantity.

currency_symbol — The string used as the local currency symbol.

mon_decimal_point — The operand is a string containing the symbol that is used as the decimal delimiter (radix character) in monetary formatted quantities. In contexts where standards (such as the ISO C standard) limit the **mon_decimal_point** to a single byte, the result of specifying a multi-byte operand is unspecified.

mon_thousands_sep The operand is a string containing the symbol that is used as a separator for groups of digits to the left of the decimal delimiter in formatted monetary quantities. In contexts where standards limit the **mon_thousands_sep** to a single byte, the result of specifying a multi-byte operand is unspecified.

mon_grouping Define the size of each group of digits in formatted monetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not –1, then the size of the previous group (if any) will be repeatedly used for the remainder of the digits. If the last integer is –1, then no further grouping will be performed.

The following is an example of the interpretation of the **mon_grouping** keyword. Assuming that the value to be formatted is **123456789** and the **mon_thousands_sep** is **’**, then the following table shows the result. The third column shows the equivalent string in the ISO C standard that would be used by the **localeconv** function to accommodate this grouping.

mon_grouping	Formatted Value	ISO C String
3;–1	123456’789	"\3\177"
3	123’456’789	"\3"
3;2;–1	1234’56’789	"\3\2\177"
3;2	12’34’56’789	"\3\2"
–1	123456789	"\177"

In these examples, the octal value of **{CHAR_MAX}** is 177.

positive_sign A string used to indicate a non-negative-valued formatted monetary quantity.

negative_sign A string used to indicate a negative-valued formatted monetary quantity.

int_frac_digits An integer representing the number of fractional digits (those to the right of the decimal delimiter) to be written in a formatted monetary quantity using **int_curr_symbol**.

frac_digits An integer representing the number of fractional digits (those to the right of the decimal delimiter) to be written in a formatted monetary quantity using **currency_symbol**.

p_cs_precedes An integer set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a monetary quantity with a non-negative value, and set to 0 if the symbol succeeds the value.

p_sep_by_space An integer set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a monetary quantity with a non-negative value, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

n_cs_precedes An integer set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a monetary quantity with a negative value, and set to 0 if the symbol succeeds the value.

n_sep_by_space An integer set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a monetary quantity with a negative value, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

p_sign_posn An integer set to a value indicating the positioning of the **positive_sign** for a monetary quantity with a non-negative value. The following integer values are recognized for both **p_sign_posn** and **n_sign_posn**:

0 Parentheses enclose the quantity and the **currency_symbol** or **int_curr_symbol**.

1 The sign string precedes the quantity and the **currency_symbol** or **int_curr_symbol**.

2 The sign string succeeds the quantity and the **currency_symbol** or **int_curr_symbol**.

3 The sign string precedes the **currency_symbol** or **int_curr_symbol**.

4 The sign string succeeds the **currency_symbol** or **int_curr_symbol**.

n_sign_posn An integer set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

The following table shows the result of various combinations:

		p_sep_by_space 2	1	0
p_cs_precedes = 1	**p_sign_posn** = 0	($1.25)	($ 1.25)	($1.25)
	p_sign_posn = 1	+ $1.25	+$ 1.25	+$1.25
	p_sign_posn = 2	$1.25 +	$ 1.25+	$1.25+
	p_sign_posn = 3	+ $1.25	+$ 1.25	+$1.25
	p_sign_posn = 4	$ +1.25	$+ 1.25	$+1.25
p_cs_precedes = 0	**p_sign_posn** = 0	(1.25 $)	(1.25 $)	(1.25$)
	p_sign_posn = 1	+1.25 $	+1.25 $	+1.25$
	p_sign_posn = 2	1.25$ +	1.25 $+	1.25$+
	p_sign_posn = 3	1.25+ $	1.25 +$	1.25+$
	p_sign_posn = 4	1.25$ +	1.25 $+	1.25$+

The monetary formatting definitions for the POSIX locale follow; the code listing depicting the **localedef**(1) input, the table representing the same information with the addition of **localeconv**(3C) and **nl_langinfo**(3C) formats. All values are unspecified in the POSIX locale.

```
LC_MONETARY
# This is the POSIX locale definition for
# the LC_MONETARY category.
#
int_curr_symbol      ""
currency_symbol      ""
mon_decimal_point    ""
mon_thousands_sep    ""
mon_grouping         -1
positive_sign        ""
negative_sign        ""
int_frac_digits      -1
p_cs_precedes        -1
p_sep_by_space       -1
n_cs_precedes        -1
n_sep_by_space       -1
p_sign_posn          -1
n_sign_posn          -1
#
END LC_MONETARY
```

The entry **n/a** indicates that the value is not available in the POSIX locale.

LC_NUMERIC The **LC_NUMERIC** category defines the rules and symbols that will be used to format non-monetary numeric information. This information is available through the **localeconv**(3C) function.

The following items are defined in this category of the locale. The item names are the keywords recognized by the **localedef** utility when defining a locale. They are also similar to the member names of the *lconv* structure defined in **<locale.h>**. The **localeconv( )** function returns **{CHAR_MAX}** for unspecified integer items and the empty string (**""**) for unspecified or size zero string items.

In a locale definition file the operands are strings. For some keywords, the strings only can contain integers. Keywords that are not provided, string values set to the empty string (**""**), or integer keywords set to –1, will be used to indicate that the value is not available in the locale. The following keywords are recognized:

decimal_point The operand is a string containing the symbol that is used as the decimal delimiter (radix character) in numeric, non-monetary formatted quantities. This keyword cannot be omitted and cannot be set to the empty string. In contexts where standards limit the **decimal_point** to a single byte, the result of specifying a multi-byte operand is unspecified.

thousands_sep The operand is a string containing the symbol that is used as a separator for groups of digits to the left of the decimal delimiter in numeric, non-monetary formatted monetary quantities. In contexts where standards limit the **thousands_sep** to a single byte, the result of specifying a multi-byte operand is unspecified.

grouping Define the size of each group of digits in formatted non-monetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not –1, then the size of the previous group (if any) will be repeatedly used for the remainder of the digits. If the last integer is –1, then no further grouping will be performed.

The non-monetary numeric formatting definitions for the POSIX locale follow; the code listing depicting the **localedef** input, the table representing the same information with the addition of **localeconv** values and **nl_langinfo** constants.

```
LC_NUMERIC
# This is the POSIX locale definition for
# the LC_NUMERIC category.
#
decimal_point      "<period>"
thousands_sep      ""
grouping           -1
#
END LC_NUMERIC
```

Item	POSIX locale Value	langinfo Constant	localeconv() Value	localedef Value
decimal_point	"."	RADIXCHAR	"."	.
thousands_sep	n/a	THOUSEP	""	""
grouping	n/a	-	""	–1

The entry **n/a** indicates that the value is not available in the POSIX locale.

LC_TIME

The **LC_TIME** category defines the interpretation of the field descriptors supported by **date**(1) and affects the behavior of the **strftime**(3C), **wcsftime**(3I), **strptime**(3C), and **nl_langinfo**(3C) functions. Because the interfaces for C-language access and locale definition differ significantly, they are described separately.

For locale definition, the following mandatory keywords are recognized:

abday Define the abbreviated weekday names, corresponding to the **%a** field descriptor (conversion specification in the **strftime( )**, **wcsftime( )**, and **strptime( )** functions). The operand consists of seven semicolon-separated strings, each surrounded by double-quotes. The first string is the abbreviated name of the day corresponding to Sunday, the second the abbreviated name of the day corresponding to Monday, and so on.

day Define the full weekday names, corresponding to the **%A** field descriptor. The operand consists of seven semicolon-separated strings, each surrounded by double-quotes. The first string is the full name of the day corresponding to Sunday, the second the full name of the day corresponding to Monday, and so on.

abmon Define the abbreviated month names, corresponding to the **%b** field descriptor. The operand consists of twelve semicolon-separated strings, each surrounded by double-quotes. The first string is the abbreviated name of the first month of the year (January), the second the abbreviated name of the second month, and so on.

mon Define the full month names, corresponding to the **%B** field descriptor. The operand consists of twelve semicolon-separated strings, each surrounded by double-quotes. The first string is the full name of the first month of the year (January), the second the full name of the second month, and so on.

d_t_fmt Define the appropriate date and time representation, corresponding to the **%c** field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain the escape sequences ****, **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**.

date_fmt Define the appropriate date and time representation, corresponding to the **%C** field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain the escape sequences ****, **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**.

d_fmt Define the appropriate date representation, corresponding to the **%x** field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain the escape sequences \\, **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**.

t_fmt Define the appropriate time representation, corresponding to the **%X** field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain the escape sequences \\, **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**.

am_pm Define the appropriate representation of the *ante meridiem* and *post meridiem* strings, corresponding to the **%p** field descriptor. The operand consists of two strings, separated by a semicolon, each surrounded by double-quotes. The first string represents the *ante meridiem* designation, the last string the *post meridiem* designation.

t_fmt_ampm Define the appropriate time representation in the 12-hour clock format with **am_pm**, corresponding to the **%r** field descriptor. The operand consists of a string and can contain any combination of characters and field descriptors. If the string is empty, the 12-hour format is not supported in the locale.

era Define how years are counted and displayed for each era in a locale. The operand consists of semicolon-separated strings. Each string is an era description segment with the format:

direction:*offset*:*start_date*:*end_date*:*era_name*:*era_format*

according to the definitions below. There can be as many era description segments as are necessary to describe the different eras.

The start of an era might not be the earliest point For example, the Christian era B.C. starts on the day before January 1, A.D. 1, and increases with earlier time.

direction Either a + or a − character. The + character indicates that years closer to the *start_date* have lower numbers than those closer to the *end_date*. The − character indicates that years closer to the *start_date* have higher numbers than those closer to the *end_date*.

offset The number of the year closest to the *start_date* in the era, corresponding to the **%Ey** field descriptor.

start_date A date in the form *yyyy*/*mm*/*dd*, where *yyyy*, *mm*, and *dd* are the year, month and day numbers respectively of the start of the era. Years prior to A.D. 1 are represented as negative numbers.

end_date The ending date of the era, in the same format as the *start_date*, or one of the two special values −* or +*. The value −* indicates that the ending date is the beginning of time. The value +* indicates that the ending date is the end of time.

era_name A string representing the name of the era, corresponding to the **%EC** field descriptor.

era_format A string for formatting the year in the era, corresponding to the **%EY** field descriptor.

era_d_fmt Define the format of the date in alternative era notation, corresponding to the **%Ex** field descriptor.

era_t_fmt Define the locale's appropriate alternative time format, corresponding to the **%EX** field descriptor.

era_d_t_fmt Define the locale's appropriate alternative date and time format, corresponding to the **%Ec** field descriptor.

alt_digits Define alternative symbols for digits, corresponding to the **%O** field descriptor modifier. The operand consists of semicolon-separated strings, each surrounded by double-quotes. The first string is the alternative symbol corresponding with zero, the second string the symbol corresponding with one, and so on. Up to 100 alternative symbol strings can be specified. The **%O** modifier indicates that the string corresponding to the value specified via the field descriptor will be used instead of the value.

LC_TIME *C-language Access*

The following information can be accessed. These correspond to constants defined in **<langinfo.h>** and used as arguments to the **nl_langinfo**(3C) function.

ABDAY_*x* The abbreviated weekday names (for example Sun), where *x* is a number from 1 to 7.

DAY_*x* The full weekday names (for example Sunday), where *x* is a number from 1 to 7.

ABMON_*x* The abbreviated month names (for example Jan), where *x* is a number from 1 to 12.

MON_*x* The full month names (for example January), where *x* is a number from 1 to 12.

D_T_FMT The appropriate date and time representation.

D_FMT The appropriate date representation.

T_FMT The appropriate time representation.

AM_STR The appropriate ante-meridiem affix.

PM_STR The appropriate post-meridiem affix.

T_FMT_AMPM The appropriate time representation in the 12-hour clock format with **AM_STR** and **PM_STR.**

ERA The era description segments, which describe how years are counted and displayed for each era in a locale. Each era description segment has the format:

direction:*offset*:*start_date*:*end_date*:*era_name*:*era_format*

according to the definitions below. There will be as many era description segments as are necessary to describe the different eras. Era description segments are separated by semicolons.

The start of an era might not be the earliest point For example, the Christian era B.C. starts on the day before January 1, A.D. 1, and increases with earlier time.

direction Either a + or a – character. The + character indicates that years closer to the start_date have lower numbers than those closer to the *end_date*. The – character indicates that years closer to the start_date have higher numbers than those closer to the *end_date*.

offset The number of the year closest to the start_date in the era.

start_date A date in the form *yyyy* / *mm* / *dd*, where *yyyy*, *mm*, and *dd* are the year, month and day numbers respectively of the start of the era. Years prior to AD 1 are represented as negative numbers.

end_date The ending date of the era, in the same format as the *start_date*, or one of the two special values –* or +*. The value –* indicates that the ending date is the beginning of time. The value +* indicates that the ending date is the end of time.

era_name The era, corresponding to the **%EC** conversion specification.

era_format The format of the year in the era, corresponding to the **%EY** conversion specification.

ERA_D_FMT The era date format.

ERA_T_FMT The locale's appropriate alternative time format, corresponding to the **%EX** field descriptor.

ERA_D_T_FMT The locale's appropriate alternative date and time format, corresponding to the **%Ec** field descriptor.

ALT_DIGITS The alternative symbols for digits, corresponding to the **%O** conversion specification modifier. The value consists of semicolon-separated symbols. The first is the alternative symbol corresponding to zero, the second is the symbol corresponding to one, and so on. Up to 100 alternative symbols may be specified.

The following table displays the correspondence between the items described above and the conversion specifiers used by **date**(1) and the **strftime**(3C), **wcsftime**(3I), and **strptime**(3C) functions.

localedef Keyword	langinfo Constant	Conversion Specifier
abday	ABDAY_*x*	**%a**
day	DAY_*x*	**%A**
abmon	ABMON_*x*	**%b**
mon	MON	**%B**
d_t_fmt	D_T_FMT	**%c**
date_fmt	DATE_FMT	**%C**
d_fmt	D_FMT	**%x**
t_fmt	T_FMT	**%X**
am_pm	AM_STR	**%p**
am_pm	PM_STR	**%p**
t_fmt_ampm	T_FMT_AMPM	**%r**
era	ERA	**%EC, %Ey, %EY**
era_d_fmt	ERA_D_FMT	**%Ex**
era_t_fmt	ERA_T_FMT	**%EX**
era_d_t_fmt	ERA_D_T_FMT	**%Ec**
alt_digits	ALT_DIGITS	**%O**

LC_TIME *General Information*

Although certain of the field descriptors in the POSIX locale (such as the name of the month) are shown with initial capital letters, this need not be the case in other locales. Programs using these fields may need to adjust the capitalization if the output is going to be used at the beginning of a sentence.

The **LC_TIME** descriptions of **abday**, **day**, **mon**, and **abmon** imply a Gregorian style calendar (7-day weeks, 12-month years, leap years, and so forth). Formatting time strings for other types of calendars is outside the scope of this document set.

As specified under **date** in **Locale Definition** and **strftime**(3C), the field descriptors corresponding to the optional keywords consist of a modifier followed by a traditional field descriptor (for instance **%Ex**). If the optional keywords are not supported by the implementation or are unspecified for the current locale, these field descriptors are treated as the traditional field descriptor. For instance, assume the following keywords:

```
alt_digits    "0th";"1st";"2nd";"3rd";"4th";"5th";\
              "6th";"7th";"8th";"9th";"10th"
d_fmt         "The %Od day of %B in %Y"
```

On 7/4/1776, the **%x** field descriptor would result in “The 4th day of July in 1776” while 7/14/1789 would come out as “The 14 day of July in 1789” It can be noted that the above example is for illustrative purposes only; the **%O** modifier is primarily intended to provide for Kanji or Hindi digits in **date** formats.

LC_MESSAGES

The **LC_MESSAGES** category defines the format and values for affirmative and negative responses.

The following keywords are recognized as part of the locale definition file. The **nl_langinfo**(3C) function accepts upper-case versions of the first four keywords.

yesexpr The operand consists of an extended regular expression (see **regex**(5)) that describes the acceptable affirmative response to a question expecting an affirmative or negative response.

noexpr The operand consists of an extended regular expression that describes the acceptable negative response to a question expecting an affirmative or negative response.

yesstr The operand consists of a fixed string (not a regular expression) that can be used by an application for composition of a message that lists an acceptable affirmative response, such as in a prompt.

nostr The operand consists of a fixed string that can be used by an application for composition of a message that lists an acceptable negative response.

The format and values for affirmative and negative responses of the POSIX locale follow; the code listing depicting the **localedef** input, the table representing the same information with the addition of **nl_langinfo( )** constants.

```
LC_MESSAGES
# This is the POSIX locale definition for
# the LC_MESSAGES category.
#
yesexpr "<circumflex><left-square-bracket><y><Y><right-square-bracket>"
#
noexpr  "<circumflex><left-square-bracket><n><N><right-square-bracket>"
#
yesstr          "yes"
nostr           "no"
END LC_MESSAGES
```

localedef Keyword	langinfo Constant	POSIX Locale Value
yesexpr	YESEXPR	"^[yY]"
noexpr	NOEXPR	"^[nN]"
yesstr	YESSTR	"yes"
nostr	NOSTR	"no"

SEE ALSO

date(1), **locale**(1), **localedef**(1), **sort**(1), **uniq**(1), **localeconv**(3C), **nl_langinfo**(3C), **setlocale**(3C), **strcoll**(3C), **strftime**(3C), **strptime**(3C), **strxfrm**(3C), **wcscoll**(3I), **wcsftime**(3I), **wcsxfrm**(3I), **charmap**(5), **regex**(5)