Internationalizing and Localizing Applications in Oracle Solaris

Part No: E54758
March 2019

Internationalizing and Localizing Applications in Oracle Solaris

Part No: E54758

Contents

Using This Documentation

- **Overview** – Describes the processes for internationalization and localization of applications in the Oracle Solaris 11.3 operating system.
- **Audience** – This guide is intended for programmers who want to create internationalized or localized applications for use with Oracle Solaris 11, with an emphasis on C programming language interfaces.
- **Required knowledge** – This guide assumes basic competence in programming and familiarity with the C programming language and the UNIX operating system. This guide also assumes basic knowledge of the internationalization and localization concepts. This guide assumes that you have the following documentation available for reference:
  - *Oracle Solaris 11.3 Programming Interfaces Guide* – Describes the Oracle Solaris 11 network and system interfaces used by application developers.
  - *International Language Environments Guide for Oracle Solaris 11.3* – Provides an overview of Oracle Solaris 11 internationalization and localization features and tools for users and system administrators.
  - *Oracle Developer Studio 12.6: C User's Guide* and *Oracle Developer Studio 12.6: C++ User's Guide*– Provides compiler documentation.
  - *The Single UNIX Specification, Version 3* – This document, also referred to as SUSv3, is the IEEE Standard (IEEE Std 1003.1-2001) and The Open Group Technical Standard that the Oracle Solaris OS implements.
  - *The Single UNIX Specification, Version 4* – This document, also referred to as SUSv4, is the latest version of the standard and the IEEE Std 1003.1-2008.
  - *The Unicode Standard, Version 6.0.0* (Mountain View, CA: The Unicode Consortium, 2011. ISBN 978-1-936213-01-6)

    http://www.unicode.org/versions/Unicode6.0.0/ – The Unicode Standard that the Oracle Solaris OS implements.
  - *CLDR - Unicode Common Locale Data Repository* – A Unicode project providing a standard repository of locale data.
  - *The Report of the IAB Character Set Workshop held 29 February - 1 March, 1996* – Provides a good description of the issues with character sets.

- *CJKV Information Processing, 2nd Edition*, by Ken Lunde, O'Reilly Media, 2008 – Provides information about Chinese, Japanese, Korean, and Vietnamese internationalization.

This guide includes many references to Oracle Solaris manual pages. The manual pages are online under Oracle Solaris 11.3 Information Library.

If you have installed the man page package on a running Oracle Solaris installation, you can use the `man` command. For example, the manual page for the `iconv` function from the Standard C Library Functions is referred to as `iconv`(3C), and can be viewed by the following command:

```
$ man -s 3C iconv
```

Product Documentation Library

Documentation and resources for this product and related products are available at `http://www.oracle.com/pls/topic/lookup?ctx=E53394-01`.

Feedback

Provide feedback about this documentation at `http://www.oracle.com/goto/docfeedback`.

♦ ♦ ♦ C H A P T E R 1

Internationalization and Localization Overview

Internationalization and localization are different procedures. *Internationalization* is the process of making software portable between languages or regions, while *localization* is the process of adapting software for specific languages or regions. Internationalized software is developed using interfaces that modify program behavior at runtime in accordance with specific cultural requirements. Localization involves establishing online information to support a language or region also called a locale.

Internationalized software works with different native languages and customs and can be ported from one locale to another without rewriting the software. The Oracle Solaris system is internationalized, providing the infrastructure and interfaces you need to create internationalized software.

Overview of Locales

A *locale* is a collection of language and cultural convention data for a specific region. In the context of a UNIX operating system such as Oracle Solaris, the term locale has a specific meaning defined by a set of standards. See the *International Language Environments Guide for Oracle Solaris 11.3* for detailed explanation.

`C` Locale

The `C` locale, also known as the `POSIX` locale, is the `POSIX` system default locale for all POSIX-compliant systems. The Oracle Solaris operating system is a `POSIX` system. The Single UNIX Specification, Version 3, defines the `C` locale. Register at `http://www.unix.org/version3/online.html` to read and download the specification.

To run the internationalized programs in the C locale use any of the following ways:

- Unset all locale environment variables. Runs the application in the `C` locale.

```
$ unset LC_ALL LANG LC_CTYPE LC_COLLATE LC_NUMERIC LC_TIME LC_MONETARY LC_MESSAGES
```

- Explicitly set the locale to `C` or `POSIX`.

```
$ LC_ALL=C
$ export LANG=C
```

Some applications check the `LANG` environment variables without actually calling `setlocale`(3C) to reference the current locale. In this case, shell is explicitly set to the `C` locale by specifying the `LC_ALL` and `LANG` locale environment variables. For the precedence relationship among locale environment variables, see the `setlocale`(3C) man page.

To check the current locale settings in a terminal environment, run the `locale`(1) command.

```
$ locale
LANG=C
LC_CTYPE="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_COLLATE="C"
LC_MONETARY="C"
LC_MESSAGES="C"
LC_ALL=
```

Locale Categories

The types of locale categories are as follows:

`LC_CTYPE`	Character classification and case conversion.
`LC_TIME`	Specifies date and time formats, including month names, days of the week, and common full and abbreviated representations.
`LC_MONETARY`	Specifies monetary formats, including the currency symbol for the locale, thousands separator, sign position, the number of fractional digits, and so forth.
`LC_NUMERIC`	Specifies the decimal delimiter (or radix character), the thousands separator, and the grouping.
`LC_COLLATE`	Specifies a collation order and regular expression definition for the locale.

`LC_MESSAGES`	Specifies the language in which the localized messages are written, and affirmative and negative responses of the locale (yes and no strings and expressions).
`LO_LTYPE`	Specifies the layout engine that provides information about language rendering. Language rendering (or text rendering) depends on the shape and direction attributes of a script.

Core Locales

The Oracle Solaris core locales are as follows:

Chinese-Simplified	`zh_CN.UTF-8`
Chinese-Traditional	`zh_TW.UTF-8`
English	`en_US.UTF-8`
French	`fr_FR.UTF-8`
German	`de_DE.UTF-8`
Italian	`it_IT.UTF-8`
Japanese	`ja_JP.UTF-8`
Korean	`ko_KR.UTF-8`
Portuguese-Brazilian	`pt_BR.UTF-8`
Spanish	`es_ES.UTF-8`

Core locales have better coverage at the level of localized messages than the locales available for additional installation. Oracle Solaris OS components such as Installer or Package Manager are localized only in core locales while localized messages for third-party software such as GNOME or Firefox are often available in more locales.

All locales in the Oracle Solaris environment are capable of displaying localized messages, provided that the localized messages for the relevant language and application are present. Additional locales including their localized messages can be added to the system from the installation repository by modifying the `pkg facet` properties.

About Internationalization

The word *internationalization* is often abbreviated as *i18n*, because there are 18 letters between the letter *i* and *n*. Although there are different definitions of what internationalization means, in this book, internationalization means making programs generic and flexible so that requirements for markets around the world are accommodated.

Part of internationalization is making the product localizable, so that the user interface (UI) of an application can be translated with minimal changes in the source code of the application.

For example, consider a message text in a C program that is coded as follows:

```
/* This is not internationalized code */
printf("This message needs internationalization.");
```

This message when externalized into a message catalog using the `gettext` command is written as follows:

```
/* This is internationalized code */
printf("%s", gettext("This message is now internationalized."));
```

Another important part of internationalization is to allow processing of data belonging to different locales without changing the source code.

For example, consider that you have to sort an array of strings. Without internationalization, the source code would be as follows:

```
/* This is not internationalized code, since strcmp() compares byte
   values, producing an invalid sort order */

if (strcmp((const char *)string[q], (const char *)string[p]) > 0) {
    temp = string[q];
    string[q] = string[p];
    string[p] = temp;
}
```

This method of sorting works if the characters to be processed are only from the English locale. However, for the code to handle characters from different locales, you must use locale-sensitive functions. Using a locale-sensitive function, the sorting method is written as follows:

```
/* This is internationalized code, since strcoll() uses locale
   information to determine sort order */

if (strcoll((const char *)string[q], (const char *)string[p]) > 0) {
    temp = string[q];
    string[q] = string[p];
    string[p] = temp;
}
```

About Localization

The word *localization* is often abbreviated as *l10n*, because there are 10 letters between the *l* and the *n*. Localization is the process of customizing an application for a particular locale. For example, customization involves the following activities:

- Translating the user interface and related documentation into a different language.
- Altering some format fields in resource files according to the locale conventions, for example, changing the date format from `mm/dd/yy` to `yy/mm/dd`.
- Adding code modules that implement locale-specific functionality, for example, an input method editor for Japanese or a module that calculates Hebrew calendar dates.

♦ ♦ ♦ C H A P T E R 2

Programming Interfaces in the `libc` Library

This chapter describes selected application programming interfaces (APIs) related to internationalization and localization, which are available in the Oracle Solaris C library (Solaris `libc`). It also provides code samples which describe the APIs.

Programming Standards

The Oracle Solaris OS supports numerous standards and specifications. The Oracle Solaris 11 release is certified and registered with the UNIX 03 brand conforming to the *Single UNIX Specification, Version 03 (SUSv3)* standard. This standard is managed by The Open Group industry consortium.

Software developers writing applications in the Oracle Solaris environment should be aware of these standards. Developers can declare that an application complies with a specific standard by using specific compiler options. The standards that are supported by Oracle Solaris and their related compiler options are outlined in the standards(5) manual page.

Managing System Locales

The locale setting of an application determines how the application behaves in different locales. The locale setting is the single most important setting when dealing with internationalized applications. From a user's perspective, the locale affects many aspects of application's behavior, and from the programmer's perspective, it affects the behavior of many interfaces that the system provides. Properly written internationalized applications need to correctly set the locale.

For more information about locales and how they affect a program's behavior, see *International Language Environments Guide for Oracle Solaris 11.3*.

Caution - All the internationalized interfaces are *MT-Safe with exceptions* as mentioned in the `setlocale`(3C) man page. For more information about interface classification, see "MT Interface Safety Levels" in *Multithreaded Programming Guide*.

Locale-Sensitive Functions

The functions described in this book are locale sensitive. The output of the functions depends on the locale of the process.

In a C program, the locale is set by using the `setlocale()` function. The `setlocale()` function must be called early in a program so that other functions can use the locale information.

Note - If you do not set any locale in your program, by default the program runs in the C locale. For more information about the C locale, see "`C` Locale" on page 9.

Locale Functions

The functions related to system locales are as follows:

`setlocale()`	Set the program locale
`localelist()`	Query installed locales
`localelistfree()`	Free memory associated with a `localelist()` call

The `localelist()` function is used to query the locales which are installed on a system. For information about how to install additional locales on an Oracle Solaris system, see *International Language Environments Guide for Oracle Solaris 11.3*.

For more information, see the `setlocale`(3C), `localelist`(3C), `localelistfree`(3C), `environ`(5), `locale_alias`(5), `langinfo.h`(3HEAD), and `nl_types.h`(3HEAD) man pages.

EXAMPLE 1 Setting the Locale of a Program

The following code fragment shows how to set the locale to `en_US.UTF-8`.

```
#include <locale.h>
```

```
      :
(void) setlocale(LC_ALL, "en_US.UTF-8");
```

Note - If you want to use the locale information from the user environment, pass an empty string ("") as an argument to the `setlocale()` function. For information, see the `setlocale`(3C) and `environ`(5) man pages.

EXAMPLE 2 Querying the Locale of a Program

The following code fragment shows how to query the current locale.

```
#include <locale.h>
      :
char *locale;
      :
locale = setlocale(LC_ALL, NULL);
```

In this example, the `locale` variable is set to the current locale of the program.

EXAMPLE 3 Using the Locale Settings From the User Environment

The following code fragment shows how to set the `env_locale` variable to use the locale settings from the user environment.

```
#include <locale.h>
      :
char *env_locale;
env_locale = setlocale(LC_ALL, "");
```

For example, if the locale in the user environment is `es_ES.UTF-8`, the `env_locale` variable is set to `es_ES.UTF-8`.

Note - When the environment has different values set for different locale categories (also called the *composite locale setting*), the call to the `setlocale` function with the `LC_ALL` category, returns a string that contains values for all the categories separated by the slash character "/". For example:

```
"/es_ES.UTF-8/es_ES.UTF-8/es_ES.UTF-8/es_ES.UTF-8/es_ES.UTF-8/de_DE.UTF-8"
```

This string includes the categories `LC_CTYPE`, `LC_NUMERIC`, `LC_TIME`, `LC_COLLATE`, `LC_MONETARY`, and `LC_MESSAGES`, where `LC_MESSAGES` was set in the environment to `de_DE.UTF-8`.

Functions for Retrieving and Formatting the Locale Data

The functions to retrieve and format the locale data are as follows:

`localeconv()`	Retrieve numeric formatting information
`nl_langinfo()`	Retrieve language and the locale information
`strftime()`	Convert date and time to a string
`strptime()`	Convert character string to a time structure
`strfmon()`	Convert monetary value to a string

These functions are used to query locale-specific data, such as the time format or currency symbol. The functions can also be used to format time, numeric, or monetary information according to regional conventions. For more information, see the `langinfo.h`(3HEAD) and `mktime`(3C) man pages.

EXAMPLE 4 Obtaining the Codeset Name of a Locale

The following code fragment shows how to obtain the `codeset` of the current program's locale.

```
#include <langinfo.h>
:
char *cs;
cs = nl_langinfo(CODESET);
```

In this example, for the C locale, the `cs` variable points to the string "*646*", which is the canonical name for the US-ASCII codeset. For more information about codesets, see "Converting Codesets" on page 29.

EXAMPLE 5 Querying the Affirmative Response String of a Locale

The following code fragment shows how to set the `yesstr` variable to the *yes/no* string, which is used for the affirmative response of the current locale.

```
#include <langinfo.h>
:
char *yesstr;
yesstr = nl_langinfo(YESSTR);
```

For example, in the `es_ES.UTF-8` locale, `yesstr` will point to the string `sí`.

EXAMPLE 6 Printing the Local Time

The following code fragment shows how to display the current date and time formatted according to the regional conventions of the locale that is set for the environment.

```
#include <stdio.h>
#include <locale.h>
#include <time.h>
   :
char    *locale, ftime[BUFSIZ];
time_t  t;

locale = setlocale(LC_ALL, "");
if (locale == NULL) {
/* handle error */
}

if (0 != strftime(ftime, BUFSIZ, (char *)NULL, localtime(&t))) {
(void) printf("%s - %s\n", locale, ftime);
}
```

Handling Messages

Any user-based application will display text information to the user, for example, menu choices, warning messages, window titles and so on. When an application is localized, the text displayed to the user must also be in the localized language.

The two sets of APIs for handling messages are described in the following sections:

- "`gettext` APIs" on page 19
- "X/Open `catgets` APIs" on page 28

`gettext` APIs

Text messages to be localized must be separated from the source code of the application and stored in a separate file. These files are referred to as *message bundles*, *message catalogs*, or *portable message files*. Every programming language provides a set of tools to work with these files. For example, in programming languages, such as the C, Python, and Perl programming languages provide `gettext` functions for translating messages.

You create portable message files with the `gettext` utility. These files are in plain text, and have `.po` as the file extension. You send the portable message file to translators for translation, and the translators update the file with the translated text. Post translation, the `.po` file contains the message ID with the corresponding translated text. For example:

```
$ cat cs.po
.
.
#: code.c:37
#,c-format
msgid "My hovercraft is full of eels.\n"
msgstr "Moje vznášedlo je plné úhořů.\n"
```

However, you can improve the performace of the system by converting a portable object file to a message object file. A message object file has `.mo` as the file extension. To convert a portable object file to a message object file, use the `msgfmt` utility.

Note - If the messages are wrapped in the `gettext` functions, translation is done depending on the current locale. This way the original text messages are used as keys to the message catalog.

▼ How to Generate Localized Message Objects for a Shell Script

1. **Prepend `/usr/gnu/bin` to your `PATH` environment variable in order to use the GNU versions of the `gettext` tools.**
2. **Extract messages from the shell script into a message file template using the `xgettext` command.**
3. **Create a portable message (`.po`) file specific to the translation language by using the `msginit` command.**
4. **Translate the created messages in the `.po` file.**
5. **Create the `LC_MESSAGES` directory in the directory specified by the `TEXTDOMAINDIR` environment variable.**
6. **Either create symbolic links or set the `LANGUAGE` variable.**
7. **Create the message object (`.mo`) file.**

Example 7 Generating Localized Message Objects for a Shell Script

This example shows how to generate localized message objects for a shell script. It assumes that you have the following shell script that has calls to the `gettext` function.

```
#!/usr/bin/bash
```

```
#

# set TEXTDOMAIN and TEXTDOMAINDIR as per the gettext(1) manual page
TEXTDOMAIN=test_gettext_sh
export TEXTDOMAIN
TEXTDOMAINDIR=/home/xxxxx/lib/locale
export TEXTDOMAINDIR
PATH=/usr/gnu/bin:/usr/bin
export PATH

# source gettext.sh for using eval_gettext and eval_ngettext
. gettext.sh

f="filename.dat"
# Use eval_gettext or eval_ngettext if it refers to shell variables

# TRANSLATORS: $f is replaced with a file name
eval_gettext "\$f not found"; echo
gettext "file not found"; echo

echo "`eval_gettext "\\\$f not found"`"
echo "`gettext "file not found"`"
```

For this shell script, you can create localized message objects by using the following steps:

1. Prepend `usr/gnu/bin` to your `PATH` environment variable in order to use the GNU versions of `gettext` tools.

   ```
   $ PATH=/usr/gnu/bin:$PATH
   ```
2. Extract messages from the shell script into a message file template using the `xgettext` command.

   ```
   $ xgettext -c"TRANSLATORS:" -L"Shell" test_gettext.sh
   ```

   A file called `messages.po` is created, which contains the header information and the message strings from the shell script. It also includes the explanatory comments for the translators. The following example shows an excerpt of the `messages.po` file:

   ```
   #. TRANSLATORS: $f is replaced with a file name
   #: test_gettext.sh:18 test_gettext.sh:21
   #, sh-format
   msgid "$f not found"
   msgstr ""

   #: test_gettext.sh:19 test_gettext.sh:22
   msgid "file not found"
   msgstr ""
   ```

3. Create a portable message (`.po`) file which is specific to the translation language by using the `msginit` command. For example, use the following command to create a `.po` file for the Japanese `ja_JP.UTF-8` locale:

   ```
   $ msginit --no-translator --locale=ja_JP.UTF-8 --input=messages.po
   ```

   A file called `ja.po` is created.
4. Translate the messages in the `ja.po` file.
5. Create the `LC_MESSAGES` directory in the directory specified by the `TEXTDOMAINDIR` environment variable.

   ```
   $ mkdir -p lib/locale/ja/LC_MESSAGES
   ```
6. Either create symbolic links or set the `LANGUAGE` variable.
   - Create symbolic links

     ```
     $ ln -s ja lib/locale/ja_JP.UTF-8
     ```
   - Set the `LANGUAGE` variable.

     ```
     $ LANGUAGE=ja_JP.UTF-8:ja
     $ export LANGUAGE
     ```
7. Create the message object (`.mo`) file.

   ```
   $ msgfmt -o lib/locale/ja/LC_MESSAGES/test_gettext_sh.mo ja.po
   ```

▼ How to Generate Localized Text Messages for a C Program

1. **Prepend `/usr/gnu/bin` to your `PATH` environment variable.**
2. **Extract messages from the source code into the message file template using the `xgettext` command.**
3. **Create another `.po` file for the `LC_TIME` locale category.**
4. **Create portable message (`.po`) files specific to the translation language by using the `msginit` command.**
5. **Translate the created `.po` files.**
6. **Create the `LC_MESSAGES` and `LC_TIME` directories in the directory specified by the `LOCALEDIR` variable.**
7. **Either create symbolic links or set the `LANGUAGE` variable.**

8. **Create the message object (`.mo`) files.**

Example 8 Generating Localized Text Messages for a C Program

This example shows how to generate localized message objects for a C program. It assumes that you have the following C program that has calls to the `gettext` function.

```
#include <stdio.h>
#include <sys/types.h>
#include <libintl.h>
#include <locale.h>
#include <time.h>



/*
 * _()   is used for the strings to extract messages.
 * N_()  is used for the string array message to extract messages.
 * T_()  is used for the strings to extract messages with working on LC_TIME
 */
#define _(String)      gettext (String)
#define gettext_noop(String)   String
#define N_(String)      gettext_noop (String)
#define T_(String)      gettext_noop (String)

#define LOCALEDIR      "/home/xxxxx/lib/locale"
#define PACKAGE         "test_gettext"

static const char *msg[] = {
   N_("The first message"),
   N_("The second message"),
};

int main(int ac, char **av)
{
   char *file = "test.dat";
   int line = 40;
   int column = 10;
   time_t tloc;
   char time_buf[BUFSIZ];

   setlocale(LC_ALL, "");
   bindtextdomain(PACKAGE, LOCALEDIR);
   textdomain(PACKAGE);
   /*
    * By default, the characters are converted to current locale's encoding.
    * If this is not desired, call bind_textdomain_codeset(). For example,
     * if you want "UTF-8" encoding, specify "UTF-8" in the second argument.
     *
     *   bind_textdomain_codeset("test_gettext", "UTF-8");
     */
    printf(_("This is a test\n"));

    printf("%s\n", _(msg[0]));
    printf("%s\n", _(msg[1]));
    /* TRANSLATORS:
       First %d is replaced by a line number.
       Second %d is replaced by a column number.
       %s is replaced by a file name. */
    printf(_("ERROR: invalid input at line %1$d, %2$d in %3$s\n"),
        line, column, file);

    /*
     * strftime() works with LC_TIME not LC_MESSAGES so to get properly
     * formatted time messages we have to call dcgettext() with LC_TIME category.
     */
    (void) time(&tloc);
    (void) strftime(time_buf, sizeof (time_buf),
        /* TRANSLATORS:
         This is time format used with strftime().
         Please modify time format to fit your locale by using
         date '+%a %b %e %H:%M:%S' */
        dcgettext(NULL, T_("%a %b %e %H:%M:%S"), LC_TIME),
        localtime(&tloc));
    printf("%s\n", time_buf);

    return(0);
}
```

For this C program, you can create localized message objects using the following steps:

1. Prepend `/usr/gnu/bin` to your `PATH` environment variable:

   ```
   $ PATH=/usr/gnu/bin:$PATH
   ```

2. Extract messages from the source code into the message file template using the `xgettext` command:

   ```
   $ xgettext -c"TRANSLATORS:" -k -k"_" -k"N_" -L"C" test_gettext.c
   ```

   The `messages.po` file is created for the `LC_MESSAGES` locale category. It contains the header information and the message strings including the explanatory comments for the translators. The following example shows an excerpt of the `messages.po` file:

   ```
   #: test_gettext.c:21
   msgid "The first message"
   ```

```
msgstr ""

#: test_gettext.c:22
msgid "The second message"
msgstr ""

#: test_gettext.c:43
#, c-format
msgid "This is a test\n"
msgstr ""

#. TRANSLATORS:
#. First %d is replaced by a line number.
#. Second %d is replaced by a column number.
#. %s is replaced by a file name.
#: test_gettext.c:51
#, c-format
msgid "ERROR: invalid input at line %1$d, %2$d in %3$s\n"
msgstr ""
```

3. Create another `.po` file for the `LC_TIME` locale category:

   ```
   $ xgettext -c"TRANSLATORS:" -k -k"T_" -L"C" -o messages_t.po test_gettext.c
   ```

   The `messages_t.po` file is created for the `LC_TIME` locale category.
4. Create portable message (`.po`) files which are specific to the translation language by using the `msginit` command.

   For example, use the following command to create a portable file messages for the Japanese `ja_JP.UTF-8` locale:

   ```
   $ msginit --no-translator --locale=ja_JP.UTF-8 \
    --input=messages.po
   $ msginit --no-translator --locale=ja_JP.UTF-8 --input=messages_t.po \
    --output-file=ja_t.po
   ```

   The `ja.po` and `ja_t.po` files are created.
5. Translate the created `ja.po` and `ja_t.po` files.
6. Create the `LC_MESSAGES` and `LC_TIME` directories in the directory specified by the `LOCALEDIR` variable.

   ```
   $ mkdir -p lib/locale/ja/LC_MESSAGES lib/locale/ja/LC_TIME
   ```
7. Either create symbolic links or set the `LANGUAGE` variable:
   - Create symbolic links.

```
$ ln -s ja lib/locale/ja_JP.UTF-8
```

- Set the `LANGUAGE` variable.

  ```
  $ LANGUAGE=ja_JP.UTF-8:ja
  $ export LANGUAGE
  ```

8. Create the message objects (`.mo` files).

   ```
   $ msgfmt -o lib/locale/ja/LC_MESSAGES/test_gettext.mo ja.po
   $ msgfmt -o lib/locale/ja/LC_TIME/test_gettext.mo ja_t.po
   ```

Message Object File Format

The message object files are created in the following format:

`/usr/lib/locale/locale/category/textdomain.mo`

The path has several components:

`/usr/lib/locale`	The default path predicate for the message object files. For example, in case of text domain collisions, the path is specified by a call to the `bindtextdomain()` function. In the case of third party software, the message object files will be available in `/usr/share/locale` directory.
locale	The locale directory.
category	The locale category.
textdomain`.mo`	The text domain specified by a `textdomain()` function call. It is a unique identifier and the file name for the message catalog.

Consider the following example:

`/usr/lib/locale/it_IT.UTF-8/LC_MESSAGES/mymessages.mo`

where:

`it_IT.UTF-8`	The locale directory. This message object contains translations for Italian language and will be used for this locale and any other which are symbolic links to this directory.
`LC_MESSAGES`	The locale category.

Note - Messages are usually in the `LC_MESSAGES` and `LC_TIME` categories.

`mymessages`	The message catalog name.

Oracle Solaris and GNU-compatible `gettext` Interfaces

The Oracle Solaris `gettext` APIs provide support for both Oracle Solaris and GNU-compatible message catalog files. However, some `gettext` APIs are specific to the GNU-compatible message catalog files. The Solaris and GNU-compatible `gettext` interfaces are as follows:

`gettext()`	Retrieve a text string from the message catalog
`dgettext()`	Retrieve a message from a message catalog for a specific domain
`textdomain()`	Set and query the current domain
`bindtextdomain()`	Bind the path for a message domain
`dcgettext()`	Retrieve a message from a message catalog for a specific domain and category

GNU `gettext` Interfaces

The `gettext` APIs that work only with GNU-compatible message catalog files are as follows:

`ngettext()`	Retrieve a text string from the message catalog and choose a plural form
`dngettext()`	Retrieve a text string from the message catalog for a specific domain and choose a plural form
`bind_textdomain_codeset()`	Specify the output `codeset` for message catalogs for a domain
`dcngettext()`	Retrieve a text string from the message catalog for a specific domain and category and choose a plural form

For more information about GNU text message handling, see the GNU gettext reference (`http://www.gnu.org/software/gettext/manual/gettext.html`).

For more information about `gettext` functions, see the `msgfmt`(1), `xgettext`(1), and `gettext`(1) man pages.

Message Handling Tools

The `gettext` provides functions and command-line tools to create and handle message object files. Oracle Solaris message objects have a different format from GNU `gettext` message objects. The Oracle Solaris variants of command-line tools to handle messages are as follows:

`/usr/bin/gettext`	Retrieve a text string from the message catalog
`/usr/bin/msgfmt`	Create a message object from a portable message file
`/usr/bin/xgettext`	Retrieve calls to `gettext` strings from C programs

The GNU variants of the command-line tools to handle messages are as follows:

`/usr/bin/ggettext`	Retrieve a text string from the message catalog
`/usr/bin/gmsgfmt`	Create a message object from a message file
`/usr/bin/gxgettext`	Retrieve `gettext` call strings

To distinguish from Oracle Solaris tools, the GNU variant tools are prefixed with the letter *g*, and are symbolic links to the `/usr/gnu/bin` directory. For example, `/usr/bin/ggettext` is a symbolic link to `/usr/gnu/bin/gettext`.

The GNU `gettext` tools are part of the `text/gnu-gettext` package, which also includes other utilities for processing message catalogs.

Note - The Python `gettext` implementation supports only the GNU `gettext` message object format. Therefore, for Python programs, you must create GNU compatible message objects.

For more information, see the `msgcat`(1), `msgcmp`(1), and `msgmerge`(1) man pages.

X/Open `catgets` APIs

The X/Open `catgets` tools and interfaces use numbers as keys to the message catalog. Therefore, you can have different translations for the same string by using an unique numeric identifier as a key. However, source code maintenance could be an issue. For example, when an English message is updated, `gettext` displays the updated English message in the localized environment until the updated translations are in place. This practice helps to identify the messages that need to be updated. But, in the case of `catgets`, the localized environment continues to display the outdated translations unless the numeric key to the message is changed along with the updated English message.

Note - In Oracle Solaris the `catgets` APIs that have been added in the X/Open standard are not commonly used. Unless compliance with the X/Open standard is required, use `gettext` APIs and tools.

The X/Open `catgets` interfaces for handling messages are as follows:

`catopen()`	Open a message catalog
`catgets()`	Read a program message
`catclose()`	Close a message catalog

The command-line tools used with the X/Open `catgets` interfaces are as follows:

`/usr/bin/gencat`	Generate a formatted message catalog
`/usr/bin/genmsg`	Extract messages from source files

Converting Codesets

In systems, characters are represented as unique scalar values. These scalar values are handled as bytes or byte sequences. The *coded character set* is the character set plus the mappings between the characters and the corresponding unique scalar values. These unique scalar values are called *codeset*. For example, 646 (also called US-ASCII) is a codeset as per the ISO/IEC 646:1991 standard for the basic Latin alphabet. The following table shows other examples of codesets:

Codeset	Character	Representation
`US-ASCII`	A	0x41
`ISO 8859-2`	Č	0xC8
`EUC-KR`	Full-width Latin A	0xA3 0xC1

The Unicode standard adds another layer and maps each character to a *code point*, which is a number between 0 and 1,114,111. This number is represented differently in each of the Unicode encoding forms, such as `UTF-8`, `UTF-16`, or `UTF-32`. For example:

Codeset	Character	Code point	Encoding	Representation
Unicode	FULLWIDTH LATIN CAPITAL LETTER A	65,313 or 0xFF21	`UTF-8`	0xEF 0xBC 0xA1
			`UTF-16LE`	0x21 0xFF

Note - A codeset is also referred to as an *encoding*. Even though, there is a distinction between these terms, *codeset* and *encoding* are used interchangeably.

Code conversion or codeset conversion means converting the byte or byte sequence representations from one codeset to another codeset. A common approach to conversion is to use the `iconv()` family of functions. Some of the terms used in the area of code conversion and `iconv()` functions are as follows:

Single-byte codeset	Codeset that maps the characters to a set of values ranging from 0 to 255, or 0x00 to 0xFF. Therefore, a character is represented in a single byte.
Multibyte codeset	Codeset that maps some or all of the characters to more than one byte.
Illegal character	Invalid character in an input codeset.
Shift sequence	Special sequence of bytes in a `multibyte codeset` that does not map to a character but instead is a means of changing the state of the decoder.
Incomplete character	Sequence of bytes that does not form a valid character in an input codeset. However, it may form a valid character on a subsequent call to the conversion function, such as the `iconv()` function, when additional bytes are provided from the input. This is common when converting a multibyte stream.
Non-identical character	Character that is valid in the input codeset but for which an identical character does not exist in the output codeset.
Non-identical conversion	Conversion of a non-identical character. Depending on the implementation and conversion options, these characters can be omitted in the output or replaced with one or more characters indicating that a non-identical conversion occurred. The Oracle Solaris `iconv()` function replaces non-identical characters with a question mark ('?') by default.

Converting Codesets by Using `iconv` Functions

The `iconv()` functions available in the `libc` library for code conversion are described as follows:

`iconv_open()`	Code conversion allocation function
`iconv()`	Code conversion function
`iconv_close()`	Code conversion deallocation function
`iconvctl()`	Control and query the code conversion behavior
`iconvstr()`	String-based code conversion function

`iconv()` functions enable the code conversion of characters or a sequence of characters from one codeset to another. The `iconv_open()` function supports various codesets. You can display information about supported codesets and their aliases currently available on a system by running the following command:

```
$ iconv -l
```

Because `iconv` modules come in multiple packages, you can extend the default list of available conversions by installing additional packages. The default installation includes the `system/library/iconv/utf-8` package, which covers the basic set of `iconv` modules for conversions among `UTF-8`, Unicode, and other selected codesets.

You can install additional packages by using the Package Manager application or the `pkg` command. If you are using the Package Manager for installation, the additional packages are available in the `System/Internationalization` category. If you are using the `pkg` command, use the `system/library/iconv/*` name pattern for installation.

The `iconv` conversion modules are in the form of `fromcode%tocode.so` and must be present in the `iconv` module library under the `/usr/lib/iconv` directory for the `iconv` functions to use them. Therefore, you cannot convert between any two codesets listed by the `iconv -l` command. When all the `iconv` packages are installed and a required module is not available, you can do a two-step conversion using a Unicode encoding, for example, `UTF-32`, as an intermediary codeset. Alternately, you can develop a custom conversion module. To create custom `iconv` conversion modules use the `geniconvtbl` utility. For information about the input file format for the `geniconvtbl` utility, see the `geniconvtbl`(4) man page.

EXAMPLE 9 Creating Conversion Descriptor Using `iconv_open()`

The following code fragment shows how to use the `iconv_open()` function for converting the string `złoty` (currency of Poland) from the single-byte ISO 8859-2 codeset to `UTF-8`. In order to perform the conversion with `iconv`, you need to create a conversion descriptor with a call to the `iconv_open()` function and verify that the call was successful.

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

```
iconv_t cd;
 :
cd = iconv_open("UTF-8", "ISO8859-2");
if (cd == (iconv_t)-1) {
   (void) fprintf(stderr, "iconv_open() failed");
    return(1);
}
```

The target codeset is the first argument to the `iconv_open()` function.

EXAMPLE 10 Conversion Using `iconv()`

The following code fragment shows how to convert one codeset to another using the `iconv()` function.

Before the actual conversion, certain variables need to be in place to hold the information returned by the `iconv` call, such as the output buffer, number of bytes left in the input and output buffers, and so on.

The L WITH STROKE character is represented as 0xB3 in hexadecimal, in the ISO 8859-2 codeset. Therefore, the input buffer (`inbuf`), which holds the input string, is set for illustrational purposes to `z\xB3oty`. The contents of `inbuf` would be a result of reading a stream or a file.

```
#include <iconv.h>
#include <stdio.h>
#include <errno.h>

    :
int        ret;
char       *inbuf;
size_t     inbytesleft, outbytesleft;
char       outbuf[BUFSIZ];
char       *outbuf_p;

inbuf = "z\xB3oty";
inbytesleft = 5;      /* the size of the input string */
```

For the output buffer to hold the converted string, at least 6 bytes are needed. The L WITH STROKE character is converted to Unicode character LATIN SMALL LETTER L WITH STROKE, represented as a two-byte sequence 0xC5 0x82 in `UTF-8`.

Because in most cases, the actual size of the resulting string is not known before the conversion, make sure to allocate the output buffer with enough extra space. The `BUFSIZ` macro defined in `stdio.h` is sufficient in this case.

```
outbytesleft = BUFSIZ;
outbuf_p = outbuf;
```

This conversion call uses the conversion descriptor `cd` from the previous example.

```
ret = iconv(cd, &inbuf, &inbytesleft, &outbuf_p, &outbytesleft);
```

After the call to `iconv`, you need to check whether it succeeded. If it was successful and there is still space in the output buffer, you need to terminate the string with a null character.

```
  if (ret != (size_t)-1) {
          if (outbytesleft == 0) {
                  /* Cannot terminate outbuf as a character string; error return */
                  return (-1);
          }
          /* success */
          *outbuf_p = '\0';
            :
  }
```

If the call is successful, the outbuf will contain the string in the `UTF-8` codeset in hexadecimal notation `\x7a\xc5\x82\x6f\x74\x79`, or `z\xc5\x82oty`. The inbuf will now point to the end of the converted string. The inbytesleft will be 0. The outbytesleft is decremented by 6, which is the number of bytes put to the output buffer. The outbuf_p points to the end of the output string in outbuf.

If the call fails, check the errno value to handle the error cases as shown in the following code fragment:

```
  if (ret != (size_t)-1)) {
          if (errno == EILSEQ) {
                  /* Input conversion stopped due to an input byte that
                   * does not belong to the input codeset.
                   */
                    :
          } else if (errno == E2BIG) {
                  /* Input conversion stopped due to lack of space in
                   * the output buffer.
                   */
                    :
          } else if (errno == EINVAL) {
                  /* Input conversion stopped due to an incomplete
                   * character or shift sequence at the end of the
                   * input buffer.
                   */
                    :
          }
  }
```

Finally, deallocate the conversion descriptor and any memory associated with it.

```
iconv_close(cd);
```

Functions for Converting Between Unicode Codesets

Functions available for converting between any two of the Unicode encoding forms `UTF-8`, `UTF-16`, and `UTF-32` are described in the following man pages:

`uconv_u8tou16`(9F)	Convert `UTF-8` string to `UTF-16`
`uconv_u8tou32`(9F)	Convert `UTF-8` string to `UTF-32`
`uconv_u16tou8`(9F)	Convert `UTF-16` string to `UTF-8`
`uconv_u16tou32`(9F)	Convert `UTF-16` string to `UTF-32`
`uconv_u32tou8`(9F)	Convert `UTF-32` string to `UTF-8`
`uconv_u32tou16`(9F)	Convert `UTF-32` string to `UTF-16`

Processing UTF-8 Strings

Functions available for processing Unicode `UTF-8` strings are described in the following man pages:

`u8_textprep_str`(9F)	String-based `UTF-8` text preparation
`u8_strcmp`(9F)	`UTF-8` string comparison function
`u8_validate`(9F)	Validate `UTF-8` characters and calculate the byte length

Note - Use the `u8_textprep_str()` function to convert a `UTF-8` string to uppercase or lowercase as well as to apply one of the Unicode normalization forms. For more information, see `http://unicode.org/reports/tr15/`.

Handling Characters and Character Strings

Character codes used for handling characters and character strings can be categorized into two groups:

Multibyte (file code)	File code is used for text data exchange and for storing in a file. It has fixed byte ordering regardless of the underlying system, which is Big Endian byte ordering. Codesets like `UTF-8`, `EUC`, single-byte codesets, `BIG5`, `Shift-JIS`, `PCK`, `GBK`, `GB18030`, and so on come under this category. The term *multibyte character* in the context of the functions described in this section is a general term that refers to the codeset of the current locale, even though it might in some cases be a single-byte codeset.
Wide characters (process code)	Process code is a fixed-width representation of a character used for internal processing. It is in the native byte ordering of the platform, which can be either Big Endian or Little Endian. Encodings like `UTF-32`, `UCS-2`, and `UCS-4` can be wide-character encodings.

Conversion between multibyte data and wide-character data is often necessary. When a program takes input from a file, the multibyte data in the file is converted into wide-character process code by using input functions like `fscanf()` and `fwscanf()` or by using conversion functions like `mbtowc()` and `mbsrtowcs()` after the input. To convert output data from wide-character format to multibyte character format, use output functions like `fwprintf()` and `fprintf()` or apply conversion functions like `wctomb()` and `wcsrtombs()` before the output.

Functions for handling characters, wide characters, and corresponding data types are described in the following sections.

Character Types and Definitions

The `ISO/IEC 9899` standard defines the term "wide character" and the `wchar_t` and `wint_t` data types.

- A wide character is a representation of a single character that fits into an object of type `wchar_t`.
- The `wchar_t` is an integer type capable of representing all characters for all supported locales.
- The `wint_t` is an integer type capable of storing any valid value of `wchar_t` or `WEOF`.
- A *wide-character string* (also *wide string* or *process code string*) is a sequence of wide characters terminated by a *null* wide character code.

Note - The `ISO/IEC 9899` standard does not specify the form or the encoding of the contents for the `wchar_t` data type. Because it is an implementation-specific data type, it is not portable. Although many implementations use some Unicode encoding forms for the contents of the `wchar_t` data type, do not assume that the contents of `wchar_t` are Unicode. Some platforms use `UCS-4` or `UCS-2` for their wide-character encoding.

In Oracle Solaris, the internal form of `wchar_t` is specific to a locale. In the Oracle Solaris Unicode locales, `wchar_t` has the `UTF-32` Unicode encoding form, and other locales have different representations.

Fore more information, see `stddef.h`(3HEAD) and `wchar.h`(3HEAD) man pages.

Integer Coded Character Classification Functions

The following functions are used for character classification and return a non-zero value for true, and `0` for false. With the exception of the `isascii()` function, all other functions are locale sensitive, specifically for the `LC_CTYPE` category of the current locale.

`isalpha()`	Test for an alphabetic character
`isalnum()`	Test for an alphanumeric character
`isascii()`	Test for a 7-bit US-ASCII character
`isblank()`	Test for a blank character
`iscntrl()`	Test for a control character
`isdigit()`	Test for a decimal digit
`isgraph()`	Test for a visible character
`islower()`	Test for a lowercase letter
`isprint()`	Test for a printable character
`ispunct()`	Test for a punctuation character
`isspace()`	Test for a white-space character
`isupper()`	Test for an uppercase letter
`isxdigit()`	Test for a hexadecimal digit

These functions should not be used in a locale with a multibyte codeset, such as `UTF-8`. Use the wide-character classification functions described in the following section for multibyte codesets.

The behavior of some of these functions also depends on the compiler options used at compile time. The `ctype`(3C) man page describes the "Default" and "Standard conforming" behaviors for `isalpha()`, `isgraph()`, `isprint()`, and `isxdigit()` functions. For example, `isalpha()` function is defined as follows:

Default `isalpha()`	Tests for any character for which `isupper()` or `islower()` is true.
Standard Conforming `isalpha()`	Tests for any character for which `isupper()` or `islower()` is true, or any character that is one of the current locale-defined set of characters for which none of `iscntrl()`, `isdigit()`, `ispunct()`, or `isspace()` is true. In the C locale, `isalpha()` returns true only for the characters for which `isupper()` or `islower()` is true.

This has consequences for languages or alphabets which have no case for its letters (also called unicase), such as Arabic, Hebrew or Thai. For alphabetic characters such as aleph (0xE0) in the Hebrew legacy locale `he_IL.ISO8859-8`, the functions `isupper()` and `islower()` always return false. Therefore, even the `isalpha()` function always returns false. If compiler options are enabled for the standard conforming behavior, the `isalpha()` function returns true for such characters. For more information, see the `isalpha`(3C) and `standards`(5) man pages.

See also the Oracle Developer Studio 12.6: C User's Guide, `ctype`(3C), and `SUSv3`(5) man pages.

Wide-Character Classification Functions

The following man pages describe functions that classify wide characters and return a non-zero value for `TRUE`, and `0` for `FALSE`. These functions check the given wide character against named character classes, such as `alpha`, `lower`, or `jkana`, which are defined in the `LC_CTYPE` category of the current locale. Therefore, these functions are locale sensitive.

`iswalpha`(3C)	Test for an alphabetic wide-character
`iswalnum`(3C)	Test for an alphanumeric wide character
`iswascii`(3C)	Test whether a wide character represents a 7-bit US-ASCII character
`iswblank`(3C)	Test for a blank wide character
`iswcntrl`(3C)	Test for a control wide character
`iswdigit`(3C)	Test for a decimal digit wide character
`iswgraph`(3C)	Test for a visible wide character

`iswlower`(3C)	Test for a lowercase letter wide character
`iswprint`(3C)	Test for a printable wide character
`iswpunct`(3C)	Test for a punctuation wide character
`iswspace`(3C)	Test for a white-space wide character
`iswupper`(3C)	Test for an uppercase letter wide character
`iswxdigit`(3C)	Test for a hexadecimal digit wide character
`isenglish`(3C)	Test for a wide character representing an English language character, excluding US-ASCII characters
`isideogram`(3C)	Test for a wide character representing an ideographic language character, excluding US-ASCII characters
`isnumber`(3C)	Test for wide character representing digit, excluding US-ASCII characters
`isphonogram`(3C)	Test for a wide character representing a phonetic language character, excluding US-ASCII characters
`isspecial`(3C)	Test for a wide character representing a special language character, excluding US-ASCII characters

The following character classes are defined in all the locales:

- `alnum`
- `alpha`
- `blank`
- `cntrl`
- `digit`
- `graph`
- `lower`
- `print`
- `punct`
- `space`
- `upper`
- `xdigit`

The isenglish(), isideogram(), isnumber(), isphonogram(), and isspecial() are legacy Oracle Solaris specific wide-character classification functions. The character classes for these functions are defined only in the following Asian locales: ko_KR.EUC, zh_CN.EUC, zh_CN.GBK, zh_CN.GB18030, zh_HK.BIG5HK, zh_TW.BIG5, and zh_TW.EUC and their variants. The return values will always be false when used in other locales including Unicode locales.

You can to query for a specific character class in a generic way by using the following functions:

wctype()	Define character class
iswctype()	Test character for specified class

EXAMPLE 11 Querying Character Class of a Wide Character

In the following example, calls to the iswctype() and wctype() functions are used to check whether the given Unicode character belongs to the jhira character class . The jhira character class is from Japanese Hiragana script.

```
wint_t  wc;
int     ret;

setlocale(LC_ALL, "ja_JP.UTF-8");

/* "\xe3\x81\xba" is UTF-8 for HIRAGANA LETTER PE */
ret = mbtowc(&wc, "\xe3\x81\xba", 3);
if (ret == (size_t)-1) {
        /* Invalid character sequence. */
        :
}

if (iswctype(wc, wctype("jhira"))) {
        wprintf(L"'%c' is a hiragana character.\n", wc);
}
```

The example will produce the following output:

ぺ is a hiragana character.

Character Transliteration Functions

The following functions serve for mapping characters between character classes (character transliteration). If a mapping for a character is in the character class of the current locale, the functions return a transliterated character. These functions are locale sensitive.

`tolower()`	Convert an uppercase character to lowercase
`toupper()`	Convert a lowercase character to uppercase
`towlower()`	Convert an uppercase wide character to lowercase
`towupper()`	Convert a lowercase wide character to uppercase

The following functions provide a generic way to perform character transliteration:

`wctrans()`	Define character mapping
`towctrans()`	Wide-character mapping

For more information about related functions for Unicode strings, see "Processing UTF-8 Strings" on page 34.

EXAMPLE 12 Transliteration of a Wide Character

The following code fragment shows how to use the `towupper()` function for transliterating a Unicode wide character to uppercase.

```
wint_t  wc;
int     ret;

setlocale(LC_ALL, "cs_CZ.UTF-8");

/* "\xc5\x99" is UTF-8 for LATIN SMALL LETTER R WITH CARON */
ret = mbtowc(&wc, "\xc5\x99", 2);
if (ret == (size_t)-1) {
        /* Invalid character sequence. */
        :
}

wprintf(L"'%c' is uppercase of '%c'.\n", towupper(wc), wc);
```

The example will produce the following output:

```
Ř is uppercase of ř.
```

String Collation

The following functions are used for string comparison based on the collation data of the current locale:

`strcoll()`	String comparison using collating information
`strxfrm()`	String transformation
`wcscoll()`, `wscoll()`	Wide-character string comparison using collating information
`wcsxfrm()`, `wsxfrm()`	Wide-character string transformation

For better performance when sorting large lists of strings, use the `strxfrm()` and `strcmp()` functions instead of the `strcoll()` function, and the `wcsxfrm()` and `wcscmp()` functions instead of the `wcscoll()` function.

When using the `strxfrm()` and `wcsxfrm()` functions, note that the format of the transformed string is not in a human-readable form. These functions are used as input to the `strcmp()` and `wcscmp()` function calls respectively.

For more information, see the `strcmp`(3C) and `wcscmp`(3C) man pages.

Conversion Between Multibyte and Wide Characters

The following functions are used for conversion between the codeset of the current locale (multibyte) and the process code (wide-character representation).

These functions are locale sensitive and depend on the `LC_CTYPE` category of the current locale. They return the same error on incomplete characters and illegal characters. For more information about illegal characters and incomplete characters, see "Converting Codesets" on page 29.

`mblen()`	Get the number of bytes in a character
`mbtowc()`	Convert a character to a wide-character code
`mbstowcs()`	Convert a character string to a wide-character string
`wctomb()`	Convert a wide-character code to a character
`wcstombs()`	Convert a wide-character string to a character string

The following functions are restartable, and can be used to handle incomplete character cases. These cases occur when an incomplete character reported from the previous call along with the additional bytes of the current call is a valid character. In order to store the state information required for this kind of processing, the functions either use a user-provided or an internal state structure of type `mbstate_t`. The `mbsinit()` function is used to detect whether an `mbstate_t` structure is in an initial state.

`mbsinit()`	Determine the conversion object status
`mbrlen()`	Get the number of bytes in a character (restartable)
`mbrtowc()`	Convert a character to a wide-character code (restartable)
`mbsrtowcs()`	Convert a character string to a wide-character string (restartable)
`wcrtomb()`	Convert a wide-character code to a character (restartable)
`wcsrtombs()`	Convert a wide-character string to a character string (restartable)

The following functions are used for conversion between the codeset of the current locale and the process code. They determine whether the integer-coded character is represented in single-byte. If not, they return EOF and WEOF respectively.

`wctob()`	Convert a wide-character to a single-byte character, if possible
`btowc()`	Convert a single-byte character to a wide character, if possible

Wide-Character Strings

The following functions are used to handle wide-character strings:

`wcslen()`, `wslen()`, `wcsnlen()`	Get length of a fixed-sized wide-character string
`wcschr()`, `wschr()`	Find the first occurrence of a wide character in a wide-character string
`wcsrchr()`, `wsrchr()`	Find the last occurrence of a wide character in a wide-character string
`wcspbrk()`	Scan a wide-character string for a wide-character code
`wcscat()`, `wscat()`, `wcsncat()`	Concatenate two wide-character strings

`wcscmp()`, `wscmp()`, `wcsncmp()`	Compare two wide-character strings
`wcscpy()`, `wscpy()`	Copy a wide-character string
`wcsncpy()`, `wsncpy()`	Copy part of a wide-character string
`wcpcpy()`, `wcpncpy()`	Copy a wide-character string, returning a pointer to its end
`wcsspn()`, `wsspn()`	Get the length of a wide-character substring
`wcscspn()`, `wscspn`	Get the length of a complementary wide-character substring
`wcstok()`, `wstok()`	Split a wide-character string into tokens
`wcsstr()`, `wscwcs()`	Find a wide-character substring
`wcwidth()`, `wcswidth()`, `wscol()`	Get the number of column positions of a wide-character or wide-character string
`wscasecmp()`, `wsncasecmp()`	Case-insensitive wide-character string comparison
`wcsdup()`, `wsdup()`	Duplicate a wide-character string

The `wcswcs()` function was marked legacy and may be removed from the `ISO/IEC 9899` standard in the future. Use `wcsstr()` function instead.

The functions for converting wide characters to numbers are as follows:

`wcstol()`, `wstol()`, `wcstoll()`, `watol()`, `watoll()`, `watoi()`	Convert a wide-character string to a long integer

`wcstoul()`, `wcstoull()`	Convert a wide-character string to an unsigned long integer
`wcstod()`, `wstod()`, `wcstof()`, `wcstold()`, `watof()`	Convert a wide-character string to a floating-point number

The following man pages describe functions that list the in-memory operations with wide characters. They are wide-character equivalents of functions like `memset()`, `memcpy()`, and so on. These functions are not affected by the locale and all `wchar_t` values are treated identically.

`wmemset`(3C)	Set wide characters in memory
`wmemcpy`(3C)	Copy wide characters in memory
`wmemmove`(3C)	Copy wide characters in memory with overlapping areas
`wmemcmp`(3C)	Compare wide characters in memory
`wmemchr`(3C)	Find a wide character in memory

Wide-Character Input and Output

The following functions are used for wide-character input and output. These functions perform implicit conversion between file code (multibyte data) and internal process code (wide-character data).

`fgetwc()`, `getwc()`	Get a wide-character code from a stream
`getwchar()`	Get a wide character from a standard input stream
`fgetws()`	Get a wide-character string from a stream
`getws()` (*)	Get a wide-character string from a standard input stream
`fputwc()`, `putwc()`	Put a wide-character code on a stream
`putwchar()`	Put a wide-character code on the standard output stream
`fputws()`	Put a wide-character string on a stream

`putws()` (*)	Put a wide-character string on the standard output stream
`fwide()`	Set the stream orientation to byte or wide-character
`ungetwc()`	Push wide-character code back into the input stream

The following functions are used for formatting wide-character input and output:

`fwprintf()`, `wprintf()`, `swprintf()`, `wsprintf()` (*)	Print formatted wide-character output
`vfwprintf()`, `vwprintf()`, `vswprintf()`	Wide-character formatted output of a `stdarg` argument list
`fwscanf()`, `wscanf()`, `swscanf()`, `wsscanf()` (*)	Convert formatted wide-character input
`vfwscanf()`, `vwscanf()`, `vswscanf()`	Convert formatted wide-character input using a `stdarg` argument list

The functions marked with (*) were added to Oracle Solaris before the UNIX 98 standard that introduced the Multibyte Support Extension (MSE). They require inclusion of the `widec.h` header instead of the default `wchar.h`.

Using Regular Expressions

The following functions are used for matching filename patterns and regular expressions:

`fnmatch()`	Match a filename or path name
`regcomp()`	Compile the regular expression
`regexec()`	Execute regular expression matching
`regerror()`	Map error codes returned to error messages
`regfree()`	Free memory allocated by the `regcomp()` function