Software Internationalization Guide

Abstract

This guide gives an overview of internationalization standards and processes. It discusses the goals of internationalization and localization, software characteristics that vary between locales, industry standards that govern internationalization, and the internationalization facilities of the HP NonStop™ Kernel Open System Services (OSS) environment.

Product Version

N.A.

Supported Release Version Updates (RVUs)

This guide supports G06.24 and all subsequent G-series release version updates until otherwise indicated by its replacement publication.

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Published</th>
</tr>
</thead>
<tbody>
<tr>
<td>526225-002</td>
<td>June 2004</td>
</tr>
<tr>
<td>Part Number</td>
<td>Product Version</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>526225-002</td>
<td>N.A.</td>
</tr>
<tr>
<td>526225-001</td>
<td>N.A.</td>
</tr>
<tr>
<td>116809</td>
<td>N.A.</td>
</tr>
<tr>
<td>096174</td>
<td>N.A.</td>
</tr>
</tbody>
</table>
1. Understanding Internationalization Concepts

Why Internationalize Software? 1-1
What Is Internationalization? 1-1
Languages, Cultures, and Code Sets 1-2
What Is Localization? 1-3
What Is a Locale? 1-4
How do You Begin Internationalizing Software? 1-4
Internationalization Standards 1-5
Introduction to the HP Internationalization Subsystem 1-5

2. Software Characteristics That Vary by Locale

Writing Systems 2-1
Latin 2-1
Arabic 2-2
East Asian 2-2
Character Sets 2-4
Code Sets 2-5
Single-Byte Code Sets 2-5
Multibyte Code Sets 2-6
Unicode 2-8
Encoding Methods 2-9
Character Data Types 2-9
Single-Byte Characters 2-10
Multibyte Characters  2-10
Wide Characters  2-10
Signed Versus Unsigned Character Data Types  2-11
Integer Versus Character Data Types  2-11
File Codes and Process Codes  2-11
Data Transparency  2-12
Character Classification  2-13
  Case Classification  2-13
  New Character Classifications  2-13
  Classification Functions  2-13
Collation  2-14
  Character-Encoded Collation  2-14
  Character-Set Collation  2-15
  Multilevel Collation  2-15
  Ideographic Character Collation  2-15
  Other Collation Considerations  2-16
Numeric Representation  2-16
  Date Formats  2-16
  Time Formats  2-17
  Numeric and Monetary Formats  2-18

3. POSIX and XPG Internationalization Model

Standards Organizations  3-1
  POSIX  3-1
  XPG  3-1
  ISO  3-1
  Uniforum  3-1

POSIX and XPG Internationalization Model  3-2

Working With Locales  3-2
  Accessing Locale Objects  3-3
  Locale Names  3-3
  Locale Environment Variables  3-3
  Locale Variable Precedence  3-4
  Setting the User Locale Environment  3-5
  Setting the Program Environment  3-6

Locale-Independent Code  3-6

Messaging System  3-7
  Creating Message Source Files  3-8
  Generating Message Catalogs  3-9
3. POSIX and XPG Internationalization Model (continued)

Messaging System (continued)
   Accessing Message Catalogs 3-10
   Messaging System Example 3-11

Code-Set Conversion 3-11
   Creating Source-Code Files of Code-Set Conversion Tables 3-12
   Generating Code-Set Conversion Tables 3-12
   Algorithmic Code-Set Converters 3-13
   Accessing Code-Set Conversion Tables 3-13
   Code-Set Conversion Example 3-14

4. The HP Internationalization Subsystem

   About the HP Internationalization Subsystem 4-1
   Components of the Internationalization Subsystem 4-1
   Supported Locales 4-1
   Supported Code-Set Converters for the TNS/R Native Environment 4-2
   Supported Code-Set Converters for the TNS Environment 4-4

   Design and Development Guidelines 4-4
   Preparing Source Code 4-4
   Locale-Sensitive Functions 4-6
   Locales in OSS Client/Server Applications 4-9
   Internationalization Functions and COBOL85 4-10
   Internationalization Functions and Guardian Multibyte Character-Set Procedures 4-11
   Internationalization Functions and HP NonStop SQL/MX 4-12
   Internationalization Functions and HP NonStop SQL/MP 4-13
   Compiling Internationalized Applications 4-14

   Testing Internationalized Applications 4-15
   Basic Testing Checklist 4-16
   General Testing Guidelines 4-16
   Testing the Application’s Use of Locales 4-17

   Troubleshooting Internationalized Applications 4-17
   Problems With the User’s Environment 4-17
   Identifying Problems in the Application 4-17
Tables (continued)

Table A-2. Software Tested For Support of Japanese and Korean Character Sets  A-3
What’s New in This Guide

Guide Information

Abstract

This guide gives an overview of internationalization standards and processes. It discusses the goals of internationalization and localization, software characteristics that vary between locales, industry standards that govern internationalization, and the internationalization facilities of the HP NonStop™ Kernel Open System Services (OSS) environment.

Product Version

N.A.

Supported Release Version Updates (RVUs)

This guide supports G06.24 and all subsequent G-series release version updates until otherwise indicated by its replacement publication.

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Published</th>
</tr>
</thead>
<tbody>
<tr>
<td>526225-002</td>
<td>June 2004</td>
</tr>
</tbody>
</table>

Document History

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Product Version</th>
<th>Published</th>
</tr>
</thead>
<tbody>
<tr>
<td>526225-002</td>
<td>N.A.</td>
<td>June 2004</td>
</tr>
<tr>
<td>526225-001</td>
<td>N.A.</td>
<td>September 2003</td>
</tr>
<tr>
<td>116809</td>
<td>N.A.</td>
<td>November 1995</td>
</tr>
<tr>
<td>096174</td>
<td>N.A.</td>
<td>January 1995</td>
</tr>
</tbody>
</table>

New and Changed Information

The following changes have occurred since the last edition:

- Information for NonStop SQL/MX has been added to Section 4, The HP Internationalization Subsystem, and minor corrections were made to information about NonStop SQL/MP.
- Information for NonStop SQL/MX has been added to Appendix A, Software Supporting Multiple Character Sets.
About This Guide

This guide provides an overview of internationalization. It includes the following topics:

- The goals of internationalization and localization
- The characteristics that vary between locales
- The industry standards governing internationalization
- The internationalization utilities
- The HP internationalization subsystem

How to Use This Guide

This guide introduces software internationalization, sometimes referred to as “I18N” or “i18n”. It gives a general overview of internationalization standards, processes, utilities, and functions. It is intended for anyone who needs a general understanding of internationalization and the HP internationalization subsystem.

Readers seeking a basic understanding of internationalization concepts and characteristics must have a moderate understanding of computer software concepts to use this guide. Developers who need full reference information to develop internationalized applications must use this guide with the books listed in Related Reading on page x.

About the Contents of This Guide

Section 1, Understanding Internationalization Concepts, defines terms associated with internationalization and localization (sometimes referred to as “L10N” or “l10n”), explains the need for internationalization, and introduces the HP internationalization subsystem of the Open System Services (OSS) environment.

Section 2, Software Characteristics That Vary by Locale, examines the characteristics of different alphabets and discusses how the characteristics affect software designs for internationalization.

Section 3, POSIX and XPG Internationalization Model, focuses on the POSIX and XPG internationalization model. It discusses setting locales; writing locale-independent code; using a messaging system; and converting between code sets.

Section 4, The HP Internationalization Subsystem, describes the HP internationalization subsystem and its components; lists supported locales and code-set converters; lists the internationalization functions; and offers guidelines for designing, developing, compiling, testing, and troubleshooting internationalized applications.

Appendix A, Software Supporting Multiple Character Sets, describes current and planned support for two-byte character sets by software used with HP NonStop servers.

The Glossary defines terms used in this guide.
Related Reading

You can find additional information in these HP manuals:

- Open System Services Shell and Utilities Reference Manual
- Common Run-Time Environment (CRE) Programmer’s Guide
- C/C++ Programmer’s Guide

Other Publications

These publications are suggested reading for more information about internationalization:

- Global Software, by Dave Taylor, published by Springer-Verlag, 1992

Notation Conventions

The subsections that follow describe:

- Hypertext Links
- General Syntax Notation
- The notation used to indicate changes within this guide

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

This requirement is described under Backup DAM Volumes and Physical Disk Drives on page 3-2.
General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

    MAXATTACH

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

    file-name

computer type. Computer type letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

    myfile.c

italic computer type. Italic computer type letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

    pathname

[ ] Brackets. Brackets enclose optional syntax items. For example:

    TERM [\system-name.]$terminal-name
    INT[ERRUPTS]

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

    FC [ num ]
    [ -num ]
    [ text ]

    K [ X | D ] address

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

    LISTOPENS PROCESS { $appl-mgr-name }
    { $process-name }

    ALLOWSU { ON | OFF }
**Vertical Line.** A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

**... Ellipsis.** An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]...
[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char..."
```

**Punctuation.** Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;
LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must type as shown. For example:

```
"[" repetition-constant-list "]"
```

**Item Spacing.** Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
$process-name.#su-name
```

**Line Spacing.** If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE
   [ , attribute-spec ]...
```

**!i and !o.** In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT ( segment-id , error !i ) ; !o
```
!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```plaintext
error := COMPRESSEDIT ( filenum ) ; !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```plaintext
error := FILENAME_COMPARE_ ( filename1:length !i:i, filename2:length ) ; !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```plaintext
error := FILE_GETINFO_ ( filenum !i, [ filename:maxlen ] ) ; !o:i
```

### Change Bar Notation

A change bar (as shown to the right of this paragraph) indicates a difference between this edition of this guide and the preceding edition. Change bars highlight new or revised information.
Understanding Internationalization Concepts

The opening of global markets and the increasing use of computers throughout the world make it essential that companies meet the needs of global business as cost-effectively as possible. Internationalizing computer applications is a key part of the HP GeoReady strategy to address global business requirements.

This section discusses the need for software internationalization, defines terms associated with internationalization, gives an overview of internationalization and localization, introduces issues associated with internationalization, and introduces the HP internationalization subsystem of the Open System Services (OSS) environment.

Why Internationalize Software?

Technology has reached nations that once did not need software, opening many opportunities for marketing software internationally. As the world’s market changes, companies must prepare to fill the needs of the global customer. Software that adapts easily to multiple languages and cultures has the greatest chance of global success, because English-based systems are increasingly unsuitable for an international audience.

If done properly, internationalizing software can reduce development and maintenance costs. A single internationalized version of the product accommodates the needs of most languages and cultures and eliminates the need for developing and maintaining numerous versions of source code and executable files.

Internationalization increases the life span of all versions of a product. It has been common practice to release a version of a product in the source country several months before releasing a localized version in target countries. A single product that is capable of accommodating a variety of users enables releasing a product to all destinations at the same time.

What Is Internationalization?

Internationalization is the process of developing or modifying software applications without making assumptions about the language, cultural data, and encoding methods that the applications are expected to handle. Internationalized applications support new languages, cultures, and code sets without requiring changes to the source code.

Most existing applications rely heavily on the language and culture of the environment in which they were developed, and do not support different languages, cultures, and code sets. An internationalized application cannot be based on culturally dependent information—it must be able to operate according to the language and customs of its users, and it must be able to operate with different code sets.
Languages, Cultures, and Code Sets

These examples describe a few differences among languages, cultures, and code sets that are encountered in international applications:

- Symbols and rules that apply to the development language are often not appropriate to a language in which the product might be used.

For example, a computer prompt that displays the phrase *yes or no* in the English language is not appropriate for Spanish-speaking users who expect to see the phrase *sí o no*. The word *sí* is not recognized as a valid word in computer implementations of the English language because the character í is not a valid character in the English alphabet.

- Currency symbols, date formats, and time formats vary by culture.

For example, the monetary symbol in the United States is the dollar sign, $. Japan’s monetary symbol that signifies the yen is ¥.

- Social expressions vary between cultures.

For example, in Western cultures the “first” name is the given name and the “last” name is the family name. In some Asian cultures, however, the family name comes first and then the given name.

- Code sets assign specific numeric values to characters within a computer. Code sets vary by geographic area, and even vary within a single geographic area.

For example, the American Standard Code for Information Interchange (ASCII) is not compatible with the Extended Binary-Coded Decimal Interchange Code (EBCDIC), a code system developed by the IBM Corporation. Applications that make no assumptions about the encoding of character data are called code-set independent applications and are illustrated by Figure 1-1.

**Figure 1-1. Internationalized Application Supporting Several Languages**
What Is Localization?

Localization is the process of adapting an application to the accepted way of presenting information in a particular culture. For an internationalized program to support a variety of languages, cultures, and code sets, the data of the source country must be transformed into data that is appropriate for the target cultures. Successful localization makes an application look as if it were designed and developed specifically for the language, culture, and code set in which it is used.

For example, an internationalized program originally developed for French users must be localized if the program is also intended for users who do not understand the French language and culture. Localizing a product includes translating text, adapting the application to cultural conventions and preferences, and possibly converting code sets.

Figure 1-2 shows an internationalized software product that was developed in English and then localized to support French, German, and Spanish locales.

Localization is more than simply translating text, because more than language is involved in adapting a product to a local culture. Colors, icons, gender references,
religious references, humor, and many more culturally dependent aspects must be adapted for each locale. For example, mailboxes around the world have a great variety of shapes, so an icon depicting an American rural mailbox is unlikely to be understood by most of the world's inhabitants.

What Is a Locale?

A locale is the part of a user's environment that defines the user's language and cultural preferences or conventions. Each locale within a computer system contains information specific to each combination of supported language, cultural data, and code set. A locale contains information such as:

- Collating sequence information, to determine the sorting order of characters defined in the locale
- Character classification information, to provide details about the type of character associated with each legal character code—for example, whether it is an alphabetic, uppercase, lowercase, punctuation, control, or space character
- Case-conversion information, to identify the possible other case of each legal character code—for example, to shift from uppercase to lowercase in languages whose alphabets include uppercase and lowercase characters
- Language information, to describe the format and setting of locale-specific cultural data—for example, date and time formats

Internationalization and localization involve more than simple geographic regions. A geographic region can consist of many different locales, each having a language, culture, and code set of its own. For example, US English differs from British English in date, time, and monetary formats. There are also territories where more than one language is spoken. For example, Switzerland has four official languages—French, German, Italian, and Romansch.

See Section 2, Software Characteristics That Vary by Locale, for more information.

How do You Begin Internationalizing Software?

The basis of internationalizing an application is the separation of a program's base function from its culturally sensitive data. Base function includes the algorithms, logic, and purpose of the program. For example, the base function of a database is storage and retrieval of information, regardless of the culture in which it is used.

Culturally sensitive data includes such elements as error messages, date formats, time formats, currency formats, collation schemes, character sets, and code sets.

Figure 1-3 on page 1-5 illustrates a program in which culturally sensitive data is isolated from the program source code.
Many internationalization standards are currently being defined. Two of the most widely recognized are the *Portable Operating System Interface* (POSIX) defined by the Institute of Electrical and Electronics Engineers (IEEE), and the *X/Open Portability Guide* (XPG) defined by X/Open. Together, these standards provide many of the guidelines followed throughout the industry.

See Section 3, POSIX and XPG Internationalization Model, for more information.

### Introduction to the HP Internationalization Subsystem

The goal of internationalization is to develop applications that can meet the needs of a global audience. The HP internationalization subsystem follows the POSIX and XPG4 standards, using a set of utilities that function according to POSIX and XPG4 specifications. See Section 3, POSIX and XPG Internationalization Model, for more information.

The HP internationalization subsystem is currently supported only on the HP Open System Services (OSS) environment. The internationalization subsystem uses a set of internationalized functions written in the C language. Programmers use the internationalization subsystem to set locales, write locale-independent code, create message catalogs, and convert between code sets.

Figure 1-4 on page 1-6 shows the components of an internationalized application.
Figure 1-4. Components of an Internationalized Application

See Section 4, The HP Internationalization Subsystem, for more information.
Software Characteristics That Vary by Locale

The primary goal of internationalization is to develop software that meets the needs of all languages and cultures. This section describes a few of the world’s writing systems, and discusses issues that must be considered for internationalization.

Writing Systems

Writing systems differ in the symbols they use, the direction in which those symbols are written and read, and the manner in which they are grouped. This section discusses three of the world’s major writing systems—Latin-based writing systems, Arabic, and East Asian.

Latin

Latin-based writing systems are block-based, as shown in Figure 2-1, and are written from left to right. Block-based writing systems consist of characters that stand alone. They include the Greek and Cyrillic alphabets as well as Latin.

Most Western European languages use the Latin writing system as a base—for example, Danish, English, French, German, Hungarian, Italian, Polish, and Spanish. Latin-based scripts have also been devised for languages that are not derived from Latin—for example, Malay, Swahili, and Vietnamese.

Figure 2-1. Block-Based Writing

Languages within the Latin-based writing system differ in some ways. Most Western European languages that are based on the Latin alphabet add characters to the alphabet, incorporating accent marks and other symbols. These marks and symbols are called diacriticals.

For example, the French alphabet, like English, begins with the character A and ends with the character Z, but contains accents and other marks that are absent from the English alphabet—for example, ç. The Danish alphabet is another example of a language that includes diacriticals—it begins with the character A and ends with the character Å.

Some languages omit characters from the Latin alphabet base—for example, Hungarian omits the characters Q and W. Some languages omit characters and add others—for example, Italian omits the characters K, W, X, and Y but includes such characters as è.
Arabic

Arabic is written from right to left, using a cursive writing style in which characters often attach to characters that precede and follow them. In Arabic, characters can have different forms at different locations within a word. For example, a character at the beginning of a word has one form, but uses a different form when embedded in the center of the word and still another form when it appears at the end of a word.

Because characters can have different forms at different positions in a word, Arabic is called a context-dependent language. Languages such as Farsi (Persian), Urdu, Pashto, Sindhi, and Kurdish use the Arabic alphabet as a base.

Figure 2-2 shows an example of cursive writing.

---

Figure 2-2. Cursive Writing

![Cursive Writing]

---

East Asian

East Asian languages typically depict a word or an idea with symbols called ideographs. An ideograph usually represents an idea, rather than a sound. Most ideographs are derived from the Chinese language. An estimated 100,000 ideographs exist, although only a subset is used frequently.

Figure 2-3 is an example of an ideograph.
Each ideographic writing system has a name. Hanzi identifies the Chinese ideographic writing systems (Traditional and Simplified Chinese); Kanji is the Japanese ideographic writing system; Hanja is the Korean ideographic writing system. Together, the Chinese, Japanese, and Korean ideographs are referred to as the Han, shown in Figure 2-4.

### Chinese

Chinese text consists exclusively of ideographs called Hanzi. Two versions of Hanzi exist—Traditional Chinese and Simplified Chinese.

Taiwan, the Republic of China, uses the complex Hanzi set known as Traditional Chinese. In the People’s Republic of China, the government has simplified some of the ideographs and eliminated others. The result is the smaller, simpler set of ideographs known as Simplified Chinese.
Japanese

Japanese ideographs are known today as Kanji. Although similar concepts can be represented in Chinese and Japanese, the two languages are linguistically different.

In addition to Kanji, the Japanese language has two phonetic systems, Katakana and Hiragana, each consisting of about 50 characters. The purpose of a phonetic character is to serve as a modifier to existing ideographs or to create meanings for new words that do not have an ideographic equivalent. Together, the two phonetic systems are called Kana. Katakana is usually used for words of a foreign origin, and Hiragana is used for words with a native origin.

Korean

Some Korean words can be written with the Hanja ideographic system, but all Korean text can be written with the phonetic writing system called Hangul. Hangul has 24 characters, each representing a specific sound. Syllables are created by combining variations between two and seven characters.

Most text processing in Korean is based on syllables. For example, an operation to make a single character bold usually makes the entire syllable bold.

Character Sets

A character set is a group of characters that is used to build the elementary units of a language. Characters in a set include letters, numbers, symbols, and others such as control characters. The English, French, Spanish, Arabic, Russian, and Danish alphabets and the Japanese Kanji are examples of character sets. Internationalized software must be character-set independent and must be able to support existing and future character sets.

Table 2-1 shows some characters in the US English, German, and Spanish character sets.

<table>
<thead>
<tr>
<th>Character Sets</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>U.S. English</td>
<td>a, b, c</td>
</tr>
<tr>
<td>German</td>
<td>ä, é, ü</td>
</tr>
<tr>
<td>Spanish</td>
<td>a, á, b, ſ</td>
</tr>
</tbody>
</table>

The characters in a character set are not directly associated with the numeric encoding that represents the character in a format that can be read by a computer. A group of encoded characters is called a code set.
Code Sets

A code set assigns a unique numeric value to each character in a character set, with a designated number of bits representing each character. In the past, hardware limitations restricted the number of bits used to represent characters, but with fewer hardware restrictions software can now support code sets of up to hundreds of thousands of characters. The expansion of hardware capabilities and the expansion of software technology into worldwide markets have resulted in the development of many code sets.

Because so many code sets exist, characters in a character set often have different values in different code sets. For example, in the American Standard Code for Information Interchange (ASCII), the character A is assigned the decimal numeric value 65 and B is assigned 66. The Extended Binary-Coded Decimal Interchange Code (EBCDIC), however, assigns the decimal value of 193 to the character A, and 194 to the character B. Internationalized software must be code-set independent to operate on different computer systems that use different code sets.

Single-Byte Code Sets

Traditionally, 7 or 8 bits have been used to represent a character and its numeric value. Figure 2-5 on page 2-5 shows how the character A is represented by 8 bits.

**Figure 2-5. Eight-Bit Representation of the Character A**

\[
A = 65 = \begin{array}{cccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

ASCII

The ASCII code set uses only 7 bits to represent the 128 characters in the code set. It contains only the uppercase and lowercase characters in the US English alphabet, a few punctuation symbols, the digits 0 to 9, and various symbols and control characters. ASCII is not appropriate for international use because of its limited number of characters.

ISO 8859

The International Organization for Standardization (ISO) 8859 series is a group of standard eight-bit code sets. ASCII is a subset of each of the ISO 8859 code sets. Eight-bit code sets can support many character sets. ISO 8859-1, the most commonly used set, supports most Western European languages. Table 2-2 shows each ISO 8859 code set and the languages it supports.
Multibyte Code Sets

Multibyte code sets represent characters that require more than one byte to store encoded values. Eight-bit code sets are often subsets of multibyte code sets.

East Asian Code Sets

The Chinese, Japanese, and Korean languages consist of several thousand ideographic characters. Eight bits are not enough to represent these characters—16 bits or more are required. The Chinese National Standard (CNS), Chinese Guo Biao (GB), Japanese Industrial Standard (JIS), and Korean Standard (KS) are groups that have created standard code sets.

Table 2-3 lists a few of the East Asian standard code sets and the languages they support.

Table 2-3. East Asian Code Sets (page 1 of 2)

<table>
<thead>
<tr>
<th>Code Set Name</th>
<th>Languages Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNS 11643</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>GB 18030</td>
<td>Modern Chinese</td>
</tr>
<tr>
<td>GB 2312</td>
<td>Simplified Chinese</td>
</tr>
</tbody>
</table>
ISO 10646 is a universal coded character set (UCS) that represents all characters and symbols from all commonly used scripts and languages. It was developed to enable processing of groups of languages that are not usually used together. For example, users in North Africa often need to use both French and Arabic, although these two code sets are not commonly processed together in other parts of the world.

ISO 10646 characters are encoded in multiple octets. An octet is an eight-bit byte in which code space is divided into four units called “group,” “plane,” “row,” and “cell.”

Figure 2-6 shows the character layout of ISO 10646.

ISO 10646 uses two basic forms for code elements:

- UCS-2, in which code elements use the two lower-order octets (row and cell). This form is also called the Basic Multilingual Plane (BMP).
- UCS-4, in which code elements use all four octets.

Composite Characters

In addition to UCS-2 and UCS-4, ISO 10646 includes an encoding method which enables combining multiple code elements to create composite sequences. This method enables combining characters to allow a very wide variety of character combinations.

The method of combining characters uses base characters, the simplest form of a character in an alphabet, with one or more combining characters to form composite character sequences.

For example, the letter á (lowercase a with an acute accent) exists in UCS-2 as a single letter with the code value 0x00 0xe1. You could also encode the letter á as a
combining character, using the base character \( a \) followed by the acute accent. As Figure 2-7 shows, the resulting code value uses four octets—two for the \( a \) (0x00 0x61) and two for the acute accent (0x03 0x01).

**Figure 2-7. Creating a Composite Character**

The combining character method of encoding allows any number of combining characters to follow a base character, thus enabling support of such languages as Arabic and Thai.

ISO 10646 has three conformance levels that provide flexibility in its implementation:

- Level 1: Combining characters not allowed.
- Level 2: Combining characters allowed only for Arabic, Hebrew, Indic, and Thai.
- Level 3: Combining characters allowed with no restrictions.

**Unicode**

The Unicode code set currently uses the two lower octets, row and cell, and does not support levels of combining characters. Unicode is identical to ISO 10646, UCS-2, Level 3, as shown in Figure 2-8 on page 2-9. Note that ASCII code 45 is not in UCS2.
Encoding Methods

Encoding methods provide a way to mix characters from different code sets so that users can use characters from multiple languages. For example, the Extended UNIX Codes (EUC) and personal computer (PC) codes mix ASCII, local phonetic, and ideographic characters in one data stream.

Extended UNIX Codes

EUC is commonly used for UNIX-based systems. EUC supports commonly mixed languages such as English and Chinese, English and Japanese, or English and Korean.

Personal Computer (PC) Codes

PC codes are typically the standard for personal computers in a given country. For example, PC codes such as Japan's Shift-JIS and China's Big 5 are popular on East Asian personal computers.

Character Data Types

A data type is part of a variable declaration that determines the kind of values a variable can represent, the operations that can be performed on a variable, and the amount of storage to allocate. Internationalization creates a demand for processing multibyte data because many computer language implementations support the
two-byte and four-byte character data types. Some character types are not defined as single or multiple byte, but can support various combinations.

**Single-Byte Characters**

A single-byte character data type consists of eight bits that represent a character. ISO 8859 characters are single-byte characters. A single byte can represent up to 256 characters.

**Multibyte Characters**

A multibyte character is a coded character that uses one or more bytes in a single data stream and that can include characters with varying widths, as shown in Figure 2-9. Multibyte characters typically consist of characters encoded in defined code sets. For example, a multibyte data stream can contain single-byte ASCII characters as well as multibyte ideographic characters. In most situations, a mechanism is needed to define the boundary between single-byte and multibyte characters—shift-in and shift-out sequences, for example.

![Figure 2-9. Multibyte Character Data Stream](VST010.vsd)

Multibyte characters are used for file codes, which are the external representations of data. A file code is the format of data that is stored on disk. See File Codes and Process Codes on page 2-11 for more information.

**Wide Characters**

A wide character is a fixed-width character wide enough to hold any coded character supported by an implementation. A wide character is an object of the wchar_t type definition, included in ISO C to enable international support.

Wide characters promote code-set independence by removing dependencies on specific code sets or encoding methods, and replacing them with general functions that can process any encoding. The wide character data type provides flexibility because it can store characters defined up to the widest character in the supported code set.

All wide characters in a single data stream are the same size. The size is defined by the implementation and is set at compile time. Wide character sizes most often used are 1, 2, or 4 bytes—8, 16, or 32 bits. For example, if wchar_t is defined as 4 bytes, all wide character data is processed in 4-byte groups, including all characters from...
Supported character sets of 1, 2, and 4 bytes. To store 1-byte and 2-byte characters, the implementation pads unused space in the 4-byte-wide character with nulls.

Figure 2-10 is an example of a 4-byte-wide character data stream. The unshaded bytes in the figure indicate null padding.

Figure 2-10. Four-Byte Wide Character Data Stream

Wide characters are not used for file codes, the external representation of data. Because wide character size varies between systems, an application might not be able to interpret wide character file codes defined on another system. Wide character types are thus used only for internal processing. For more information about file codes and internal processing codes, see File Codes and Process Codes on page 2-11.

Signed Versus Unsigned Character Data Types

ASCII-based programs do not need to differentiate between signed and unsigned characters for most purposes, but programs using larger code sets often do. Large code sets require eight bits or more to fully represent characters, so it must be possible to distinguish between signed and unsigned characters without using the eighth bit for the sign. Internationalized software must use an unsigned character data type to store characters, so that the character can never be misinterpreted as a negative binary value.

Integer Versus Character Data Types

When an application compares integer data types to character data types it might produce unexpected results if multiple bytes are used for both character and integer representation. For example, consider a conditional statement that compares the variable \( b \) to the end-of-file (EOF) character:

```c
int b;
if ((b = getchar()) == EOF)
```

Because the return value of `getchar()` is an integer, it is important that the return value be stored as an integer also so that the comparison will work properly.

File Codes and Process Codes

A file code is the bit pattern representing a character in a character set. It is the external representation of data and is the format of data stored on disk.
A process code is an internal representation of data and is the format in which programs process data. With process codes all characters are represented internally by the same number of bits so that processing is independent of the character set. Process code is never saved on disk or exchanged with other running processes.

File codes and process codes must be considered when internationalizing applications because different character data types are used depending on whether data is stored or processed. Conversion between different character data types is thus often needed.

When processing multibyte character data types it is usually most efficient to convert to wide character types because multibyte characters can be varying widths, whereas wide characters are all a single fixed width. Programmers can make no assumptions about process code encoding because multibyte characters may not have the same encoding in wide character form.

Another method of processing multibyte character data is to process it as plain single-byte characters, so that a program can examine each byte of the multibyte data to find the character boundaries.

Data that is single-byte char-based, or that processes multibyte data without interpreting its characters, can be used as both file codes and process codes. Data that is wchar_t-based, however, can only be used as process codes. wchar_t-based data is usually defined by the implementation to accommodate the largest character in the character set. Because character size varies between implementations, wchar_t process code is not portable across systems; data stored by one system could be misinterpreted by another system trying to read the data.

**Data Transparency**

Data transparent software processes 8-bit characters without modifying or using any part of the characters in a way that is inconsistent with the rules of the code set. Data transparency is a fundamental requirement for internationalized applications.

In most single-byte code sets, all eight bits are used for encoding characters; in multibyte code sets, more than eight bits are needed. Internationalized program code must therefore examine all eight bits of every data byte and not ignore one or more bits. A program that avoids changing any data bits and that examines all eight bits of a data byte is called data transparent, or eight-bit clean.

Many programs developed in the United States use the American Standard Code for Information Interchange (ASCII). Because ASCII can be encoded with 7 bits, the eighth bit of a byte has often been used as a utility bit—for example, for parity checking or to store processing control codes—instead of as a data bit.

An internationalized program cannot depend on a single code set, nor can it use any bit in a data byte for its own purposes—it must be able to support numerous single-byte or multibyte code sets.
Character Classification

Character classification is the grouping of characters into named classes that share an attribute associated with the name of the class. For example, ASCII character classes are uppercase, lowercase, alphabet, digit, and punctuation. It is easiest to determine how to process characters if the classification of a character is defined. Internationalization requires additional character classifications to accommodate rules and symbols beyond those included in the ASCII character set.

Case Classification

Classifying characters by uppercase or lowercase does not satisfy all international character sets. Many writing systems such as Arabic, Hindi, and Japanese have character sets that are not differentiated by case. Languages that do differentiate by case contain exceptions. In German, for example, the lowercase character ß has no single-character, uppercase equivalent. Instead, the character ß is converted to the two uppercase characters SS.

New Character Classifications

Most programming languages do not allow classifying character sets for languages that are not Latin-based. The phonetic and ideographic writing systems are two examples of systems that do not classify characters. For languages like Hindi and Thai, classes must differentiate between vowels and consonants.

Classification Functions

Programs often process characters based on their character classification groups. Internationalized character classification functions are locale-dependent and help programmers avoid hard-coding characters that belong to a given class. Some programming languages provide classification features to support internationalization.

The C programming language, for example, includes the isalpha() function to determine if a character is a valid alphabetic character. Instead of comparing a character to hard-coded characters in the ASCII code set, a program calls isalpha() to determine if the character belongs to the alphabet class appropriate to the current locale. The isalpha() function can then be used to determine if a character belongs to the alphabet class of a new locale.

For example, in a program enabled for the US English locale, isalpha() returns false to verify that the character í is not a valid member of the US English alphabet. If the same program is enabled for Spanish, isalpha() returns true because the character í is a valid member of the Spanish alphabet.

The C programming language also has routines that perform class conversions. For example, tolower() converts uppercase characters to lowercase characters. Most existing C-type functions, however, provide support only for the ASCII code set.
Class-conversion routines might be written, for example, for cases in which each letter has only one uppercase version and one lowercase version. In French, however, lowercase letters may lose their diacriticals when converted to uppercase—$e$, è, é, and ë may all convert to $E$. To meet international needs, locales give users the option of defining uppercase and lowercase mappings so that diacriticals are not lost.

## Collation

Collation is the logical ordering of characters based on defined precedence algorithms. Collation algorithms vary from one language to another and can be based on character sets, character encoding values, user-defined ordering, or numerous other factors. Internationalized software must support a large variety of collation algorithms to accommodate all existing and future written languages.

### Character-Encoded Collation

A frequently used collation method is based on the encoded values of a character set. Table 2-4 illustrates an ASCII-encoded collation scheme.

<table>
<thead>
<tr>
<th>ASCII Encoded Values</th>
<th>Before Collation</th>
<th>After Collation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=65</td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>B=66</td>
<td>F</td>
<td>B</td>
</tr>
<tr>
<td>C=67</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>D=68</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>E=69</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>F=70</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

For characters in the order D, F, C, A, B, E, the result after collating by increasing ASCII-encoded values is A, B, C, D, E, F. However, the encoded collation scheme used for the ASCII character set is inappropriate for most other character sets. Collating ASCII characters by their encoded values works well because the ASCII character set is encoded in order; this is not the case for most character sets. The encoded values of uppercase characters in the ISO 8859-1 character set are not in any particular order.

Potential problems exist with collation based on character encoding. If a code set is used for more than one language, as in the case of ISO 10646, collating by encoded values is difficult because the same character is likely to appear in various positions depending on the language. For example, the character ä appears in the beginning of the German alphabet, but it appears at the end of the Swedish alphabet.

Some languages require multiple collation passes. In ASCII, encoding of all uppercase characters is positioned before encoding of all lowercase characters, but there might
be instances in which an uppercase character is followed by its lowercase counterpart instead of the next uppercase character. For example, instead of the traditional A, B, C ..., a, b, c order, the appropriate collation scheme might be A, a, B, b, C, c, ..., Z, z.

**Character-Set Collation**

Character-set collation schemes are based on the actual character instead of the encoded values, resolving some problems of character-encoded collation. With character-set collation, all existing and new character sets can be collated appropriately, independent of encoded values.

With this approach, a number of different collation orders can be defined for a single character set. Character sets that are case insensitive can have collation orders in which the uppercase and lowercase versions of a single character have the same sort value. Punctuation, symbols, and word hyphenators can be defined with the rest of the character set.

**Multilevel Collation**

With multilevel collation, several collation passes are made to refine collations. Collation that involves case-sensitive characters and diacritics often requires multilevel collation passes. In Spanish, for example, characters with the same base character (with or without diacritics) are weighed equally during collation.

*Table 2-5* shows the results of a multilevel collation based on the Spanish character set collation scheme. In the first collation pass, characters are grouped according to the base character without consideration for the diacritical. The characters a and á are therefore weighed equally in the first pass and the words mas and más collate the same. Because the two words collate the same, a second collation pass is made on them. The second pass recognizes the diacritical above the base character a so that the character a sorts first, followed by the character á.

<table>
<thead>
<tr>
<th>Words to Collate</th>
<th>Result of Collation</th>
</tr>
</thead>
<tbody>
<tr>
<td>masacrar</td>
<td>mas</td>
</tr>
<tr>
<td>mas</td>
<td>más</td>
</tr>
<tr>
<td>máscara</td>
<td>masa</td>
</tr>
<tr>
<td>más</td>
<td>masacrar</td>
</tr>
<tr>
<td>masa</td>
<td>máscara</td>
</tr>
</tbody>
</table>

**Ideographic Character Collation**

Ideographic writing systems are composed of several thousand characters. Collation methods in ideographic writing systems are more complex than those used for phonetic systems, and can be based on various factors. Generally, a collation scheme based on a combination of stroke count, radical base, and phonetics is used.
**Stroke Count**

One approach to collating ideographic characters is based on the number of strokes that make up the character. Characters containing fewer strokes sort first, followed by characters with more strokes.

**Radical Base**

Ideographic characters can be collated using a scheme based on radicals, which are the root structure of ideographs.

**Phonetics**

Pronunciation is another way of collating ideographic characters. A collation method based on pronunciation is the most difficult approach for collating ideographs because it is difficult to show how one element relates to a neighboring element.

**Other Collation Considerations**

When collating characters, you can define characters that are given no weight. These characters are called “don’t-care characters.” If a hyphen is defined as a don’t-care character, for example, the words *re-creation* and *recreation* collate to the same position.

In *n*-to-one character mappings, a string of characters is treated as a single collating element. An example is the Spanish character *ch* that appears between *c* and *d* when collated. There are also one-to-*n* character mappings where a single collating element is mapped to a string of characters. For example, the German character *ß* collates as *ss*.

**Numeric Representation**

Date formats, time formats, and monetary figures are represented in many ways around the world, so internationalized software must be flexible. Internationalized software must provide a way to overcome fixed numeric representation for date, time, and monetary formats.

**Date Formats**

Date formats vary among countries and cultures. A date consists of the year, the month, and the day in a variety of orders of presentation.

*Table 2-6* on page 2-17 shows how some languages represent dates.
Software Characteristics That Vary by Locale

Time Formats

Time formats have many variations. Some languages format time based on a 12-hour clock, such as US English, Australian English, and British English. Languages such as

---

**Table 2-6. Date Formats by Language**

<table>
<thead>
<tr>
<th>Format</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm/dd/yy</td>
<td>Australian English</td>
</tr>
<tr>
<td></td>
<td>British English</td>
</tr>
<tr>
<td></td>
<td>Canadian French</td>
</tr>
<tr>
<td></td>
<td>Danish</td>
</tr>
<tr>
<td></td>
<td>Spanish</td>
</tr>
<tr>
<td>dd-mm-yy</td>
<td>Dutch</td>
</tr>
<tr>
<td></td>
<td>Flemish</td>
</tr>
<tr>
<td></td>
<td>Italian</td>
</tr>
<tr>
<td></td>
<td>Norwegian</td>
</tr>
<tr>
<td>dd.mm.yy</td>
<td>Finnish</td>
</tr>
<tr>
<td></td>
<td>French</td>
</tr>
<tr>
<td></td>
<td>German</td>
</tr>
<tr>
<td></td>
<td>Swiss French</td>
</tr>
<tr>
<td></td>
<td>Swiss German</td>
</tr>
<tr>
<td>yy/mm/dd</td>
<td>Portuguese</td>
</tr>
<tr>
<td>yy-mm-dd</td>
<td>Swedish</td>
</tr>
</tbody>
</table>

Some countries capitalize month and day names; others do not. **Table 2-7** shows how one date is represented in a variety of ways, both in different countries and within the same country.

---

**Table 2-7. Date Formats by Country**

<table>
<thead>
<tr>
<th>Country</th>
<th>Date Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denmark</td>
<td>fredag, 20, marts 1994</td>
</tr>
<tr>
<td>England</td>
<td>Friday, 20 March 1994</td>
</tr>
<tr>
<td></td>
<td>20/3/94</td>
</tr>
<tr>
<td>France</td>
<td>vendredi, le 20 mars 1994</td>
</tr>
<tr>
<td></td>
<td>ven, 20 mars 1994</td>
</tr>
<tr>
<td></td>
<td>20.3.94</td>
</tr>
<tr>
<td>Italy</td>
<td>venerdi 20 marzo 1994</td>
</tr>
<tr>
<td></td>
<td>20-III-94</td>
</tr>
<tr>
<td>Japan</td>
<td>94.3.20</td>
</tr>
<tr>
<td>United States</td>
<td>Friday, March 20, 1994</td>
</tr>
<tr>
<td></td>
<td>Fri., Mar. 20, 1994</td>
</tr>
<tr>
<td></td>
<td>3/20/94</td>
</tr>
<tr>
<td></td>
<td>3-20-94</td>
</tr>
</tbody>
</table>
Canadian French, Danish, Dutch, Flemish, German, Italian, Portuguese, and Swiss French commonly use the 24-hour clock format.

Table 2-8 shows written time formats used by different countries.

<table>
<thead>
<tr>
<th>Country</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>France</td>
<td>16h10</td>
</tr>
<tr>
<td>Germany</td>
<td>16.10</td>
</tr>
<tr>
<td>Japan</td>
<td>16:10</td>
</tr>
<tr>
<td>United States</td>
<td>4:10 p.m.</td>
</tr>
</tbody>
</table>

To complicate time formats further, the world is divided into 24 time zones, each with its own name. Some zones have more than one name, and some names represent more than one time zone. In addition, some countries or regions within a country move their clocks forward or back an hour at times during the year to maximize daylight, while others do not.

**Numeric and Monetary Formats**

Presentation of numeric and monetary values varies from country to country. The United States and England use periods to denote decimal parts and commas to separate thousands, but most other countries reverse these definitions.

Table 2-9 shows numeric separators used in various countries.

<table>
<thead>
<tr>
<th>Country</th>
<th>Numeric Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finland</td>
<td>1 234.56 (space used for thousands separator; period as decimal character)</td>
</tr>
<tr>
<td>Germany</td>
<td>1,234,56 (period used as thousands separator; comma as decimal character)</td>
</tr>
<tr>
<td>United States</td>
<td>1,234.56 (comma used as thousands separator; period as decimal character)</td>
</tr>
</tbody>
</table>

Various symbols, alphabetic characters, and combinations of the two are used to define currency. Some countries use the currency symbol at the beginning of the quantity, some use it at the end of the quantity, and others embed the symbol within the quantity.

Table 2-10 shows monetary formats used by various countries.
### Table 2-10. Monetary Formats by Country

<table>
<thead>
<tr>
<th>Country</th>
<th>Currency</th>
<th>Monetary Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Japan</td>
<td>yen</td>
<td>¥1,234</td>
</tr>
<tr>
<td>Norway</td>
<td>krona</td>
<td>kr1.234$56</td>
</tr>
<tr>
<td>Portugal</td>
<td>escudos</td>
<td>1.234$56</td>
</tr>
<tr>
<td>Switzerland</td>
<td>Swiss francs</td>
<td>1.234$56SFrs</td>
</tr>
<tr>
<td>United States</td>
<td>dollars</td>
<td>$1,234.56</td>
</tr>
</tbody>
</table>
POSIX and XPG Internationalization Model

This section gives a general overview of the POSIX and XPG internationalization model. The scope of the POSIX and XPG standards is much greater than the information covered in this publication—for details on these standards, see the POSIX and XPG guidelines.

Standards Organizations

The computer industry has formed organizations to create and help maintain international standards for operating systems and programming languages to address the increasing demand for data portability and consistency. Organizations participating in this effort include POSIX, XPG, ISO, and Uniforum.

POSIX

The Portable Operating System Interface (POSIX) is a standard developed by a group of committees (called “dot groups”) formed by the Institute of Electrical and Electronics Engineers (IEEE). Each dot group concentrates on producing specifications for a particular aspect of the system. These specifications are usually referred to as “POSIX.n,” where n represents the dot group. The POSIX.1 and POSIX.2 specifications contain the most information about internationalization.

XPG

X/Open is an industry consortium that creates system specifications. Companies that are members of X/Open agree to conform to the specifications, which are available in a series of publications named the X/Open Portability Guide (XPG). Each release number is appended to the XPG abbreviation; for example, release 4 of the X/Open Portability Guide is called the XPG4 specification.

ISO

The International Organization for Standardization (ISO) promotes national standards to international standards. ISO contributed a series of eight-bit code sets, ISO 8859, that supports Latin, Cyrillic, and many other Western European alphabet scripts. ISO is also responsible for contributing the universal code set ISO 10646, which includes characters for most modern written languages in the world.

Uniforum

Uniforum is an international association of open systems professionals that contributes to various standards. As a technical resource for both formal and informal standards
development bodies, Uniforum’s technical committee on internationalization provides specifications for POSIX and XPG standards.

**POSIX and XPG Internationalization Model**

POSIX and XPG standards provide a model for developing internationalized software in which a user specifies a locale for the operating environment. At run time, an internationalized application inherits the specified locale information and interacts with the user in the specified locale.

The POSIX and XPG model consists of these key elements:

- Initialization of a program’s locale using environment variables
- Development of locale-independent code, using functions that support international behavior to replace hard-coded, locale-specific behavior
- A system that retrieves message text that has been separated from program code
- A method that converts a program’s data from one code set to another code set

Figure 3-1 shows the key components of the POSIX and XPG model that make up an internationalized application program.

**Figure 3-1. The POSIX and XPG Internationalization Model**

![Diagram of the POSIX and XPG Internationalization Model]

The HP internationalization subsystem includes proprietary extensions to standard internationalization functions, to address distributed internationalization in a NonStop server’s homogeneous networked environment.

**Working With Locales**

A locale is the part of a user’s environment that contains rules representing language and cultural conventions for a specific language or region. A locale consists of one or more categories. Each category controls locale-specific components of the system—
for example, collation procedures and date and time formats. Locale-specific information is isolated from a program’s source code and stored in separate locale source files.

Isolating locale-specific information in separate locale source files simplifies localizing a program. When an internationalized program needs to support a new locale, only the locale source file is localized. It is not necessary to modify or recompile the program’s source code for every new locale.

**Accessing Locale Objects**

A program must be properly initialized before functions that support international behavior can access locale-specific information. The `setlocale()` function links a program with appropriate locale data. For details about `setlocale()`, see the `setlocale(3)` reference page either online or in the *Open System Services Library Calls Reference Manual*.

**Locale Names**

XPG defines a locale name as a language, territory, and code set combination that is specified by language and territory codes.

The format for a locale name is:

```
Language[_Territory][.Codeset][@modifier]
```

- **Language** specifies a lowercase abbreviation for a language name from ISO 639; for example, `en` for English and `ja` for Japan.
- **Territory** specifies an uppercase territory abbreviation from ISO 3166; for example, `US` for the United States and `JP` for Japan.
- **Codeset** specifies the name of a code set or encoding method; for example, `Shift-JIS`.
- **@modifier** selects a specific instance of localization data within a single category; `@modifier` is not supported in the HP internationalization subsystem.

The HP internationalization subsystem uses full locale names, fully specifying language, territory, and code set—for example, `ja_JP.SJIS`. See Section 4, The HP Internationalization Subsystem, for more information.

**Locale Environment Variables**

The POSIX and XPG standards define the set of locale environment variables available in the OSS environment. These variables are a subset of a system’s environment variables.

The locale name assigned to the locale environment variables enables an application to establish the characteristics of a user’s environment after `setlocale()` is called. For example, assigning the German locale name to the `LC_TIME` locale variable
enables the user’s environment to display time formats according to German time conventions, after `setlocale()` has successfully set the German locale.

See Setting the Program Environment on page 3-6 for more information.

The following locale environment variables are available in the OSS environment:

- **LC_ALL** determines values of all locale environment variables and has precedence over all other locale environment variables. LC_ALL identifies overall language and cultural requirements and can be used alone when a single set of localization data covers all aspects of native language.

- **LANG** determines values of all locale environment variables in the absence of other LC variables. The locale setting of LANG determines how character sets, character types, collation sequences, messages, and date and time formats work. LANG can be used with specific locale variables to provide operation in multiple languages.

- **LC_COLLATE** determines how characters are ordered, sorted, grouped, and translated for specific language and cultural requirements of the user’s environment.

- **LC_CTYPE** determines character handling procedures, interpretations of bytes per character in a data stream, classification of characters, and the behavior of character classes for specific language and cultural requirements of the user’s environment.

- **LC_MESSAGES** determines how to format messages and process interactive responses for specific language and cultural requirements of the user’s environment.

- **LC_MONETARY** determines how to format representations of currencies, including symbols for monetary values for specific language and cultural requirements of the user’s environment.

- **LC_NUMERIC** determines how numbers are represented, including the use of numeric punctuations such as decimal points and commas, for specific language and cultural requirements of the user’s environment.

- **LC_TIME** determines date and time formatting for specific language and cultural requirements of the user’s environment.

The `locale` command can be used from the OSS shell to display information about available public locales, specified locale categories, and the names and values of specified locale keywords. See the `locale(1)` reference page either online or in the Open System Services Shell and Utilities Reference Manual for more information.

### Locale Variable Precedence

The locale environment variables follow these precedence rules:

1. **LC_ALL** takes precedence over all other LC variables and the LANG variable.
2. **LC variables** take precedence when LC_ALL is not defined.
3. **LANG** takes precedence when neither **LC_ALL** nor the **LC** variables are defined.

4. If **LC_ALL**, the **LC** variables, and **LANG** are all undefined, the default locale (the **C/POSIX** locale) is used.

In the following example, all aspects of the French locale (**fr_FR**) are supported except collation, which is based on German (**de_DE**) conventions:

```
LANG=fr_FR.ISO8859-1
LC_COLLATE=de_DE.ISO8859-1
```

In the next example, however, the precedence of **LC_ALL** overrules German collation:

```
LC_ALL=fr_FR.ISO8859-1
LC_COLLATE=de_DE.ISO8859-1
```

The assignment to **LC_COLLATE** has no effect because **LC_ALL** takes precedence over all other international environment variables.

### Setting the User Locale Environment

A user’s environment can inherit the **C/POSIX** default locale or be set to a different locale. The user’s environment is set either from the user’s start-up file or from the OSS shell. For the environment settings to take effect the locale must be exported, whether from a start-up file or from the shell.

With a start-up file, the environment settings are retained until the user modifies the start-up file. At the shell level, environment settings remain intact until the user exits the session or resets the values. If a locale is not defined, the user’s environment defaults to the **C/POSIX** locale.

### Examples: Setting the Locale from a Start-Up File

In this example, a section of a start-up file sets the user environment to interact with the user in US English but to display the date and time formats in German:

```
export LANG=en_US.ISO8859-1
export LC_TIME=de_DE.ISO8859-1
```

In this example, a section of a start-up file sets the user environment to interact with the user in French but to collate according to the German dictionary:

```
export LANG=fr_FR.ISO8859-1
export LC_COLLATE=de_DE.ISO8859-1
```

### Example: Setting the Locale From the Shell

This example uses the shell to set the locale first to US English, then to Spanish, and then to French:

```
$export LANG=en_US.ISO8859-1
$export LC_ALL=es_ES.ISO8859-1
```
Setting the Program Environment

An application program can inherit the user’s current locale or take on an independent locale, depending on the setting of `setlocale()`. Internationalized programs cannot function as they were designed to function until `setlocale()` is successfully called, establishing the locale.

When called in the first line of code in the main program, the `setlocale()` function initializes locale categories. `setlocale()` uses the locale specified in the locale parameter. If the locale parameter is the null string, `setlocale()` uses the environment variables. At run time, the application program can switch to any locale the system supports.

A truly internationalized application calls `setlocale()` with the null string locale (""), allowing the software to adapt to the user’s current locale. This example of a `setlocale()` call uses the null string for a locale name:

```c
/*Program that accepts the user’s current locale setting*/
main()
{
    setlocale(LC_ALL,"");
    ...
}
```

This example enables the application for the `LC_ALL` variable:

```c
/*Program that is set to the French locale*/
main()
{
    setlocale(LC_ALL,"fr_FR.ISO8859-1");
    ...
}
```

The C locale, also known as the POSIX locale, specifies the behavior of American-based systems without internationalization. If `setlocale()` is not called, the C/POSIX locale is used as the default.

Locale-Independent Code

Internationalized functions are POSIX/XPG-compliant, locale-sensitive calls that access language and cultural data. Internationalized functions eliminate hard-coding locale-sensitive data and enable software to support new locales easily.

Internationalized functions are part of the C run-time library. These functions make it possible to change locales and produce valid results according to the locale setting.
This example uses hard-coded values to check that a character is within the boundaries of the ASCII code set:

```c
/* Hard code ASCII range */
main()
{
    int input;

    input = getchar();
    if ((input >= 65 && input <= 90) || (input >= 97 && input <= 122))
        process_valid_char(input);
    else
        process_invalid_char(input);
}
```

This example uses the internationalized function `isalpha()` to verify that a character is within the boundaries of a code set defined by the current locale setting:

```c
/* Use isalpha() function */
main()
{
    int input;

    setlocale(LC_ALL,"");

    input = getchar();
    if (isalpha(input))
        process_valid_char(input);
    else
        process_invalid_char(input);
}
```

The second example provides much more flexibility than the first because the locale can change and the program still functions as it should. It contains no hard-coded, locale-dependent values as does the first example.

## Messaging System

An international messaging system enables creating, storing, and accessing user-visible text in several languages. A messaging system helps the localization process by isolating message text from program source code and storing the text in separate message source files.

Programmers create message source files, then use a utility to create message catalogs from the message source files. The message catalogs can then be localized into different languages without affecting program source code—it is not necessary to modify or recompile the program source code because of changes to message text. At run time, the program accesses the appropriate localized message catalog through a set of internationalized function calls.

Message source files are used as input to three utilities: `mkcatdefs`, `gencat`, and `runcat`. The `mkcatdefs` utility preprocesses a message text source file to change...
symbolic identifiers into numeric constants, then produces a set of commands suitable for passing to the `gencat` utility.

`gencat` creates and modifies a message catalog from a message text source file. `runcat` runs `mkcatdefs` and sends its output to `gencat`.

Two other messaging system utilities are available. The `dspcat` utility displays all or part of a message catalog; the `dspmsg` utility writes a selected message to standard output.

For more information about these messaging system utilities, see Generating Message Catalogs on page 3-9 and the associated reference pages either online or in the Open System Services Shell and Utilities Reference Manual.

Creating Message Source Files

To create message source files, programmers separate user-visible text from the program source code and store it in one or more message source files. User-visible text includes messages, errors, warnings, help text, command prompts, and responses to prompts. Isolating messages beginning in the early stages of development avoids later rework to extract message text from the source code.

Example: Source Code That is not Internationalized

This example shows the source code of a program that has not been internationalized:

```c
/*Existing program source code that has not been internationalized*/
/*Use of hard-coded message text*/
#include <stdio.h>
main()
{
  char response;

  printf("Main Menu
");
  printf("1 - Add Record
");
  printf("2 - Delete Record
");
  printf("3 - Modify Record
");
  printf("4 - Quit

");
  printf("Make A Selection:");
  scanf("%c", &response);
  switch (response)
  {
    case '1': add_record_module();
              break;
    case '2': delete_record_module();
              break;
    case '3': modify_record_module();
              break;
    case '4': break;
    default: printf("You pressed an invalid key. Try Again\n");
  }
}
```
Example: Message Source File From an Internationalized Program

This is an example of a message source file in US English that contains program messages in a format defined by XPG. This example is based on the program used in the previous example:

```bash
$ English message source file
$set 1 main module
1 "Main Menu"
2 "1 - Add Record"
3 "2 - Delete Record"
4 "3 - Modify Record"
5 "4 - Quit"
6 "Make A Selection:"
7 "You pressed an invalid key. Try Again"
$set 2 add_record_module
1 "1 - Add Record Number"
2 "2 - Enter First Name"
3 "3 - Enter Last Name"
4 "4 - Enter Phone Number"
$set 3 delete_record_module
1 "1 - Delete Record Number"
$set 4 modify_record_module
1 "1 - Modify Record Number"
2 "2 - Modify First Name"
3 "3 - Modify Last Name"
4 "4 - Modify Phone Number"
```

This program can accommodate other locales easily, because the text in this file can be localized for each supported locale—the source code does not need to be revised for each locale.

Generating Message Catalogs

A message catalog is typically a direct-access storage area containing program messages. Message catalogs are generated using the XPG-specified `gencat` utility.

The input to the `gencat` utility is one or more message source files, or output from the `mkcatdefs` utility. `mkcatdefs` preprocesses a message source file to change symbolic identifiers into numeric constants, then produces a set of commands suitable for passing to `gencat`. The `runcat` utility can be used to create a message catalog if the `mkcatdefs` and `gencat` utilities can be run using no options.

The output of the `gencat` utility is a message catalog that contains all the message data in a machine-readable format and offers data security by preventing unwanted changes to the message data.

Figure 3-2 on page 3-10 shows the generation of a Spanish and an English message catalog.

See the associated reference pages either online or in the Open System Services Shell and Utilities Reference Manual for more information about these utilities.
Accessing Message Catalogs

An internationalized program must be able to retrieve message text from a message catalog. Three XPG-defined message catalog functions access localized messages:

- **catopen()** opens a message catalog. The current locale setting determines the catalog to be opened, based on the message catalog name.
- **catgets()** retrieves messages from the opened message catalog and returns the localized messages to the calling program.
- **catclose()** closes the message catalog.

The locale environment variables define the current locale setting and are used by **catopen()** to determine which message catalog to open. The environment variable
**NLSPATH** identifies the search path to the appropriate directory for finding the message catalog; it also identifies message catalog naming conventions.

### Messaging System Example

The following example shows code modified to incorporate the XPG messaging system. It contains calls that access messages in a message catalog. A default message string is defined in the fourth parameter of the `catgets()` function and is used if the message cannot be located.

```c
/*Program source code using the XPG messaging system*/
#include <stdio.h>
#include <locale.h>
#include <nl_types.h>
main()
{
  char response;
  nl_catd catd;

  setlocale(LC_ALL,"");
  catd = catopen("EngFile.cat",NL_CAT_LOCALE);

  printf("%s",catgets(catd,1,1,"Main Menu"));
  printf("%s",catgets(catd,1,2,"1 - Add Record\n"));
  printf("%s",catgets(catd,1,3,"2 - Delete Record\n"));
  printf("%s",catgets(catd,1,4,"3 - Modify Record\n"));
  printf("%s",catgets(catd,1,5,"4 - Quit\n\n\n\n"));
  printf("%s",catgets(catd,1,6,"Make A Selection:"));
  scanf("%c",&response);
  switch (response)
  {
    case '1':   add_record_module();
       break;
    case '2':   delete_record_module();
       break;
    case '3':   modify_record_module();
       break;
    case '4':  break;
    default:   printf("%s",catgets(catd,1,7,"You pressed an invalid key. Try again\n"));
  }
  catclose(catd);
}
```

### Code-Set Conversion

Because many codes sets exist, characters are often encoded in different ways on different systems. Code-set conversion converts a character set from one encoding to another encoding.

Code-set conversion converts from one code set to another code set by using algorithms or tables. The `genxlt` utility takes as input two different code sets and builds a code-set conversion table. The code-set conversion table is then accessed by internationalized programs using the `iconv()` function.
Creating Source-Code Files of Code-Set Conversion Tables

A source-code file of code-set conversion tables contains the original code set values and their corresponding target code set values. Here is an excerpt from a source file of code-set conversion tables to convert the IBM 850 code set to the ISO 8859-1 code set:

<table>
<thead>
<tr>
<th>IBM 850</th>
<th>ISO 8859-1</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80</td>
<td>0xc7</td>
<td>cedilla</td>
</tr>
<tr>
<td>0x81</td>
<td>0xfc</td>
<td>diaeresis</td>
</tr>
<tr>
<td>0x82</td>
<td>0xe9</td>
<td>acute</td>
</tr>
<tr>
<td>0x83</td>
<td>0xe2</td>
<td>circumflex</td>
</tr>
<tr>
<td>0x84</td>
<td>0xe4</td>
<td>diaeresis</td>
</tr>
</tbody>
</table>

The first column contains entries from the IBM 850 code set. The second column contains values that correspond to the ISO 8859-1 code set. The third column is a comment specifying the character that is mapped to the values.

Generating Code-Set Conversion Tables

The `genxlt` utility is used to generate code-set conversion tables for most single-byte and multibyte code sets. Input to `genxlt` is the source-code file of code-set conversion tables. The output of `genxlt` is the code-set conversion table.

The naming convention for the code-set conversion table is the concatenation of the source code-set name to the target code-set name. For example:

```
IBM-850_ISO8859-1
```

is the name of the code-set conversion table when the input to the `genxlt` utility is the source-code file of code-set conversion tables for defining conversion of the IBM 850 code set to the ISO 8859-1 code set.

Figure 3-3 on page 3-13 shows the generation of a code-set conversion table.

See the `genxlt(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual* for more information about the utility.
Most multibyte code sets, such as the large Asian code sets, cannot use tables for conversion and therefore require algorithmic code-set converters. The naming convention for code-set converters is the same as for conversion tables. For example, the converter that defines the conversion of Shift-JIS to the Japanese EUC is called SJIS_AJEC.

Accessing Code-Set Conversion-Tables

The following XPG-defined conversion functions and the `iconv` command available in the OSS shell provide access to code-set converters or conversion tables:

- `iconv_open()` opens a code-set converter or a code-set conversion table based on the search path information contained in the `LOCPATH` environment variable. If `LOCPATH` is not defined, `iconv_open()` uses the default value of `/usr/lib/nls/loc/iconvTable`.

- `iconv()` performs the conversion from one code set to another code set.

- `iconv_close()` closes a code-set conversion table or code-set converter.

See the associated reference pages either online or in the *Open System Services Library Calls Reference Manual* for more information about these code-set conversion functions.

The `iconv` command converts the encoding of characters in a file from one coded character set to another and writes the results to the standard output file. See the `iconv(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual* for more information.
Code-Set Conversion Example

This is an example of code-set conversion:

/* Example of code-set conversion*/
#include <stdio.h>
#include <iconv.h>

main()
{
iconv_t cd;
char **inbuf, **outbuf;
size_t *inbytesleft, *outbytesleft;
size_t result;

/* Allocate memory for inbuf, outbuf, inbytesleft and outbytesleft */
cd = iconv_open ("IBM-850", "ISO8859-1");

/* Before calling iconv(), inbuf contains a string of characters for conversion from one code set to another; inbytesleft indicates the number of bytes in inbuf; outbytesleft indicates the number of available bytes in outbuf. */
result = iconv(cd, inbuf, inbytesleft, outbuf, outbytesleft);

/* After calling iconv(), outbuf contains the converted data; inbuf pointer, outbuf pointer, inbytesleft and outbytesleft are updated */
iconv_close(cd);
}
This section describes the HP internationalization subsystem, lists supported locales, gives design and development guidelines for internationalized software, and offers tips for testing and troubleshooting internationalized software.

About the HP Internationalization Subsystem

Internationalizing software is a key part of the HP GeoReady strategy to address global business requirements. The HP internationalization subsystem is used in the Open System Services (OSS) environment to develop internationalized applications. Programmers use the internationalization subsystem to set locales, write locale-independent code, create message catalogs, and perform code-set conversion.

The HP internationalization subsystem provides:

- Internationalization utilities that comply with XPG4 standards.
- Internationalization functions that comply with XPG4 standards. These functions are available in the Open System Services (OSS) C run-time library.
- An internationalization function extension for the OSS environment that changes the locale of a server process to match the locale of a client.

Components of the Internationalization Subsystem

The HP internationalization components enable development of internationalized applications in the OSS environment. Programmers use the OSS C run-time library to develop internationalized applications. This library contains the internationalization functions. Programmers use the `c89` utility to compile and link internationalized applications.

The C/POSIX locale is the default locale, and is available to all programs. Additional locales can be accessed at run time by internationalized applications. All supported single-byte and multibyte locale data and methods (located in T8372) are available to customers who purchase run-time support for all locales.

Supported Locales

HP offers locales for internationalized applications to access at run time. An internationalized application uses the C/POSIX locale by default. See Compiling Internationalized Applications on page 4-14 for more information.
Single-byte Locales

These locales are included in T8372:

da_DK.ISO8859-1 (Danish)
de_CH.ISO8859-1 (German, Switzerland)
de_DE.ISO8859-1 (German, Germany)
el_GR.ISO8859-7 (Greek)
en_GB.ISO8859-1 (English, Great Britain)
en_JP.ISO8859-1 (Japanese-English, Japan)
en_US.ISO8859-1 (English, USA)
es_ES.ISO8859-1 (Spanish)
fi FI.ISO8859-1 (Finnish)
fr_BE.ISO8859-1 (French, Belgium)
fr_CA.ISO8859-1 (French, Canada)
fr_CH.ISO8859-1 (French, Switzerland)
fr_FR.ISO8859-1 (French, France)
is_IS.ISO8859-1 (Icelandic)
it_IT.ISO8859-1 (Italian)
nl_BE.ISO8859-1 (Dutch-Belgian, Belgium)
nl_NL.ISO8859-1 (Dutch, Netherlands)
no_NO.ISO8859-1 (Norwegian)
pt_PT.ISO8859-1 (Portuguese)
sv_SE.ISO8859-1 (Swedish)
tr_TR.ISO8859-9 (Turkish)

Multibyte Locales

The 21 single-byte locales listed previously and the following four locales comprise T8372; all 25 locales are shipped to customers who order run-time support for all locales.

ja_JP.AJEC (Japanese, EUC)
ja_JP.SJIS (Japanese, SJIS)
ko_KR.eucKR (Korean, EUC)
zh_TW.eucTW (Taiwanese, EUC)

Supported Code-Set Converters for the TNS/R Native Environment

The HP internationalization subsystem supports the algorithmic and table-driven code-set converters listed in this section for the TNS/R native environment. The native environment supports many more code-set converters than does the TNS environment.

Algorithmic Converters for TNS/R

eucJP ↔ SJIS
SJIS ↔ eucJP
FSS-UTF ↔ ISO8859-1
FSS-UTF ↔ ISO8859-2
FSS-UTF ↔ ISO8859-3
### Supported Code-Set Converters for the TNS/R Native Environment

<table>
<thead>
<tr>
<th>Converter 1</th>
<th>Converter 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSS-UTF</td>
<td>ISO8859-4</td>
</tr>
<tr>
<td>FSS-UTF</td>
<td>ISO8859-5</td>
</tr>
<tr>
<td>FSS-UTF</td>
<td>ISO8859-6</td>
</tr>
<tr>
<td>FSS-UTF</td>
<td>ISO8859-7</td>
</tr>
<tr>
<td>FSS-UTF</td>
<td>ISO8859-8</td>
</tr>
<tr>
<td>FSS-UTF</td>
<td>ISO8859-9</td>
</tr>
<tr>
<td>FSS-UTF</td>
<td>SJIS</td>
</tr>
<tr>
<td>FSS-UTF</td>
<td>UCS-2</td>
</tr>
<tr>
<td>FSS-UTF</td>
<td>eucJP</td>
</tr>
<tr>
<td>FSS-UTF</td>
<td>eucKR</td>
</tr>
<tr>
<td>FSS-UTF</td>
<td>eucTW</td>
</tr>
<tr>
<td>ISO8859-1</td>
<td>UCS-2</td>
</tr>
<tr>
<td>ISO8859-2</td>
<td>UCS-2</td>
</tr>
<tr>
<td>ISO8859-3</td>
<td>UCS-2</td>
</tr>
<tr>
<td>ISO8859-4</td>
<td>UCS-2</td>
</tr>
<tr>
<td>ISO8859-5</td>
<td>UCS-2</td>
</tr>
<tr>
<td>ISO8859-6</td>
<td>UCS-2</td>
</tr>
<tr>
<td>ISO8859-7</td>
<td>UCS-2</td>
</tr>
<tr>
<td>ISO8859-8</td>
<td>UCS-2</td>
</tr>
<tr>
<td>ISO8859-9</td>
<td>UCS-2</td>
</tr>
<tr>
<td>SJIS</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>SJIS</td>
<td>UCS-2</td>
</tr>
<tr>
<td>UCS-2</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>UCS-2</td>
<td>ISO8859-1</td>
</tr>
<tr>
<td>UCS-2</td>
<td>ISO8859-2</td>
</tr>
<tr>
<td>UCS-2</td>
<td>ISO8859-3</td>
</tr>
<tr>
<td>UCS-2</td>
<td>ISO8859-4</td>
</tr>
<tr>
<td>UCS-2</td>
<td>ISO8859-5</td>
</tr>
<tr>
<td>UCS-2</td>
<td>ISO8859-6</td>
</tr>
<tr>
<td>UCS-2</td>
<td>ISO8859-7</td>
</tr>
<tr>
<td>UCS-2</td>
<td>ISO8859-8</td>
</tr>
<tr>
<td>UCS-2</td>
<td>ISO8859-9</td>
</tr>
<tr>
<td>eucJP</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>eucJP</td>
<td>UCS-2</td>
</tr>
<tr>
<td>eucKR</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>eucKR</td>
<td>UCS-2</td>
</tr>
<tr>
<td>eucTW</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>eucTW</td>
<td>UCS-2</td>
</tr>
</tbody>
</table>

### Table-Driven Converters for TNS/R

<table>
<thead>
<tr>
<th>Converter 1</th>
<th>Converter 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-850</td>
<td>ISO8859-1</td>
</tr>
<tr>
<td>IBM-857</td>
<td>ISO8859-9</td>
</tr>
<tr>
<td>IBM-869</td>
<td>ISO8859-7</td>
</tr>
<tr>
<td>ISO8859-1</td>
<td>IBM-850</td>
</tr>
<tr>
<td>ISO8859-7</td>
<td>IBM-869</td>
</tr>
<tr>
<td>ISO8859-9</td>
<td>IBM-857</td>
</tr>
<tr>
<td>ISO8859-1</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>ISO8859-2</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>ISO8859-3</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>ISO8859-4</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>ISO8859-5</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>ISO8859-6</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>ISO8859-7</td>
<td>FSS-UTF</td>
</tr>
<tr>
<td>ISO8859-8</td>
<td>FSS-UTF</td>
</tr>
</tbody>
</table>
Supported Code-Set Converters for the TNS Environment

The HP internationalization subsystem supports these code-set converters in the TNS environment:

ISO8859-9 ↔ FSS-UTF
UCS-2 ↔ SJIS
UCS-2 ↔ eucJP
UCS-2 ↔ eucKR
UCS-2 ↔ eucTW

Design and Development Guidelines

Software developers can minimize the effort necessary to internationalize applications by following internationalization guidelines from the beginning stages of development. Successfully internationalizing applications using the HP internationalization subsystem in the OSS environment depends on attention to these factors:

- **Code preparation**: Proper code preparation is the basis for implementing the POSIX and XPG4 internationalization standards. See Preparing Source Code on page 4-4 for code preparation guidelines.

- **POSIX and XPG4 internationalization standards**: These standards enable internationalization of new and existing software in the OSS environment. See Section 3, POSIX and XPG Internationalization Model, for more information.

- **Common Run-Time Environment (CRE)-compliance**: CRE compliance is required for non-C program modules bound into programs written for the OSS environment. Such programs must be developed and compiled for CRE execution to be compatible with the C code that supports the internationalization functions. See the Common Run-Time Environment (CRE) Programmer’s Guide.


Preparing Source Code

An existing program’s source code usually requires some preparation before you can apply the POSIX and XPG4 internationalization standards. This section gives guidelines for code preparation—ensuring data transparency, removing hard-coded messages, removing culturally dependent information, and removing assumptions about character encoding. To minimize the internationalization effort, incorporate these guidelines into new programs beginning in the early stages of development.
Data Transparency

Make sure your program source code is data transparent. In data transparent code, no data bits is used to represent a non-data character.

Data transparency is a fundamental requirement for internationalized applications because internationalized software must be able to support numerous single-byte or multibyte code sets. In most single-byte code sets, all eight bits are used for encoding characters; in multibyte code sets, more than eight bits are needed.

Do not use bit masks to set or test for specific bits. For example, do not use 0x80 in the eighth bit as a flag.

Hard-Coded Messages

Remove all hard-coded messages from the program source code and store them in a separate file or files. Replace the hard-coded messages with function calls to retrieve messages from the message file.

Here is an example of a hard-coded message:

```c
printf("Hello World");
```

Instead of hard-coding messages, use the messaging system. This example uses the `catgets()` function to retrieve a message from a message catalog:

```c
/* default string is included in function call */
printf("%s", catgets(catd,1,1,"Hello World"));
```

If no localized message is available, the application uses the fourth parameter of the `catgets()` function as the default message.

Culturally Dependent Information

Culturally dependent information reflects a particular language or culture and is specific to a locale. Examples of culturally dependent information include date, time, and monetary formats. Internationalizing software requires removing all hard-coded culturally dependent information from the program source code.

Here is an example of a hard-coded date format:

```c
printf("The date is %s/%s/%s", month, day, year);
```

Instead of hard-coding the date format, isolate locale-specific date formats from the program source code, and use internationalized functions to write locale-independent code. This example displays the current day:

```c
char * dptr;

strptime (dptr, MAXSIZE, "%A", tptr);
printf ("%s %s", catgets (catd,1,1,"Today is "), dptr);
```
The application takes the localized “Today is” phrase from the message catalog, and strftime() provides the day. If the locale is English, the preceding example displays “Today is Thursday”; the Italian locale results in “Oggi è giovedì”; the Spanish locale results in “Hoy es jueves.” If no localized message is available, the application uses the fourth parameter of the catgets() function as the default message.

**Character Encoding**

Remove all assumptions about character encoding from program source code. This example checks if a character is alphabetic, but assumes that ASCII encoding has been used:

```c
if ((C >= 65 && C <= 90) || (C >= 97 && C <= 122))
    printf("Valid Entry");
```

Instead of making encoding assumptions, use internationalized functions such as isalpha() to write locale-independent code:

```c
if (isalpha(c))
    printf("%s", catgets(catd,1,1,"Valid Entry");
```

**Locale-Sensitive Functions**

Internationalized functions enable character-based data to be processed independently of the underlying character encoding. Internationalized functions are standard C language functions that have been enhanced to handle languages and code sets other than US English and the ASCII coded character set. Internationalized functions are locale-sensitive, which means that their results vary at run time depending upon the current locale of the application.

**ISO/ANSI C Functions**

The ISO C standard includes functions that operate in an international environment. The following table lists these functions and identifies those whose behavior is locale sensitive (LS), based on the target compilation environment. The compilation environment is set with the SYSTYPE pragma.

---

**Note.** Locale sensitivity is available only for the OSS environment. This includes program modules compiled for the Guardian environment and bound into an OSS program.

<table>
<thead>
<tr>
<th>ISO/ANSI C function</th>
<th>OSS and Guardian TNS/R native module</th>
<th>OSS TNS module</th>
<th>Guardian TNS module</th>
</tr>
</thead>
<tbody>
<tr>
<td>atof()</td>
<td>LS</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>atoi()</td>
<td>LS</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>atol()</td>
<td>LS</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>ISO/ANSI C function</td>
<td>OSS and Guardian TNS/R native module</td>
<td>OSS TNS module</td>
<td>Guardian TNS module</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>fprintf()</td>
<td>LS</td>
<td>LS</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>fscanf()</td>
<td>LS</td>
<td>LS</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>isalnum()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>isalpha()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>iscntrl()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>isdigit()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>isgraph()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>islower()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>isprint()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>ispunct()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>isspace()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>isupper()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>isxdigit()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>localeconv()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>mblen()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>mbstowcs()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>mbtowc()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>perror()</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>printf()</td>
<td>LS</td>
<td>LS</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>scanf()</td>
<td>LS</td>
<td>LS</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>setlocale()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>sprintf()</td>
<td>LS</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>sscanf()</td>
<td>LS</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>ISO/ANSI C function</td>
<td>OSS and Guardian TNS/R native module</td>
<td>OSS TNS module</td>
<td>Guardian TNS module</td>
</tr>
<tr>
<td>strftime()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>strtod()</td>
<td>LS</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>strerror()</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>strtol()</td>
<td>LS</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>strtoul()</td>
<td>LS</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>strftime()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
<tr>
<td>strftime()</td>
<td>LS</td>
<td>C/POSIX only</td>
<td>C/POSIX only</td>
</tr>
<tr>
<td>strxfrm()</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
</tbody>
</table>
Locale-Sensitive Functions

Additional XPG4 Functions

XPG4 defines functions in addition to those listed under ISO/ANSI C Functions on page 4-6. The following functions are used for handling messages, converting code sets, processing locale-dependent date and time strings, converting monetary values, and reading the current setting of cultural data items:

- catclose()
- catgets()
- catopen()
- iconv()
- iconv_close()
- icnv_open()
- regfree()
- nl_langinfo()
- regcomp()
- regexec()
- regerror()
- structmon()
- strptime()
- wcstombs()
- wcsncmp()
- wcsncmp()
- wcsncpy()
- wcspbrk()
- towlower()
- wcscat()
- wcschr()
- wcscpy()
- wcscspn()
- wcscspn()
- wcsstrchr()
- wcstod()
- wcstok()
- wcstol()
- wcstoul()
- wcstwidth()
- wcsxfrm()
- wcsftime()
- wctype()

Worldwide Portability Functions

XPG4 defines this set of functions that are equivalent to the standard ISO C functions but that work on wide-character values:

- fgetwc()
- fgetws()
- fputwc()
- fputws()
- getwc()
- getwchar()
- iswctype()
- iswalnum()
- iswalnum()
- iswalpha()
- iswcntrl()
- iswctrl()
- iswctrl()
- iswgraph()
- iswgraph()
- iswlower()
- iswpunct()
- iswspace()
Locales in OSS Client/Server Applications

For homogeneous client/server applications in the OSS environment, HP provides the setlocale_from_msg() function to enable a server to receive a client's locale information along with messages from $RECEIVE.

If a client that is not internationalized communicates with an internationalized server, the only locale available is the C/POSIX default locale. If an internationalized client communicates with a server that is not internationalized, the result is undefined.

To change a server's locale to match the locale associated with a client's message, the server must specify that it wants to receive locale information:

- Use FILE_OPEN_ option <13> (Locale Support) when opening $RECEIVE. This option specifies that data messages are to have locale information sent with them, and enables setlocale_from_msg() to adopt the locale of a client. After doing a READUPDATE[X] operation on the $RECEIVE file, the server can adopt the locale of a client from information sent with a write or writeread message from the client.
- When the server processes a client's request, call setlocale_from_msg() to change the server locale to match that of the client before executing any other locale-sensitive function.

See the setlocale_from_msg(3) reference page or the Open System Services Library Calls Reference Manual for more information.

Example: Changing a Server Locale to Match the Client Locale

The following example sets and saves the existing locale environment, then sets the locale based on the received message, performs some operations in that locale, and restores the locale to the saved environment. A program typically uses setlocale() to set its locale environment to the locale specified by the user's environment variables and uses setlocale_from_msg() to set the locale to match that of the client.

```c
#include <locale.h>
#include <string.h>
#include <cextdecs.h>

void Do_setlocale_from_message(void)
{
    char   *client_lc, *server_lc;
    short  rf_num, options, read_cnt, fn_len;
    char  r_buf[80], receive_info[17];

    /* Setting the locale from the environment variable LC_ALL. */
    server_lc = setlocale(LC_ALL,"");
    options = 4U;  /* set bit-13 on so that locale information can be received. */
    FILE_OPEN_="$RECEIVE", 8, &rf_num,,,1, options);
```
READUPDATE(rf_num, r_buf, &read_cnt);
/* Retrieve the message tag */
FILE_GETRECEIVEINFO_(receive_info);

/* Change locale based on the received message's locale */
if (setlocale_from_msg(receive_info[2]) != NULL)
{
    /* Perform operations in the locale received from msg */
}
/* Restore the server's locale */
server_lc = setlocale(LC_ALL, server_lc);

**Internationalization Functions and COBOL85**

C functions can be called from COBOL programs. However, HP COBOL is not locale-sensitive and is not extended for use of the OSS internationalization functions. For more information about COBOL85 internationalization features, see the *COBOL85 for NonStop Systems Manual*.

**Character Sets and Collating Sequences**

The COBOL compilers do not support all character sets supported by OSS.

Only the ASCII and EBCDIC single-byte character sets are directly supported; user-defined character code sets can be defined through the ALPHABET clause. National literals and PICTURE character strings for national data items allow support for non-Roman character sets such as the Japanese Kanji alphabet in alphanumeric data; national literals contain two bytes per character but do not directly correspond to multibyte character sets as defined for OSS locales.

HP COBOL compilers support alphabet names for the following Roman-derived native character sets in the Environment Division’s Configuration Section SPECIAL-NAMES paragraph:

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Alphabets</th>
</tr>
</thead>
<tbody>
<tr>
<td>DANSK-NORSK</td>
<td>Danish, Norwegian</td>
</tr>
<tr>
<td>DEUTSCH</td>
<td>German</td>
</tr>
<tr>
<td>ESPANOL</td>
<td>Spanish</td>
</tr>
<tr>
<td>FRANCAIS-AZ</td>
<td>French (AZERTY keyboard)</td>
</tr>
<tr>
<td>FRANCAIS-QW</td>
<td>French (QWERTY keyboard)</td>
</tr>
<tr>
<td>SVENSK-SUOMI</td>
<td>Swedish, Finnish</td>
</tr>
<tr>
<td>UK</td>
<td>United Kingdom</td>
</tr>
<tr>
<td>USASCII</td>
<td>United States ASCII</td>
</tr>
</tbody>
</table>

These alphabet names can be used in the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph, the COLLATING SEQUENCE phrase.
of a SORT or MERGE statement, and the CODE-SET clause of a file description entry to provide control over behaviors similar to those provided through OSS locales.

The CURRENCY SIGN and DECIMAL POINT clauses of the SPECIAL-NAMES paragraph allow formatting behaviors similar to those provided through OSS locales.

**Date-Time Formats**

COBOL date and time formats are not affected by locale. Data Division language features provide the only format controls for both kinds of data.

**Message Text**

COBOL compiler and run-time diagnostic messages do not vary by locale.

**Internationalization Functions and Guardian Multibyte Character-Set Procedures**

Guardian procedures can be called from OSS processes that also use OSS internationalization functions. Such Guardian procedure calls are not locale-sensitive.

Guardian processes can use the set of Guardian procedure calls that have names prefixed by MBCS_ to normalize multibyte character data for use with an application written to support only ASCII data. See the Guardian Programmer’s Guide for an overview of these multibyte character-set procedure calls and the Guardian Procedure Calls Reference Manual for the specific actions supported.

HP recommends against mixing the Guardian multibyte character-set procedures and the OSS internationalization functions in the same application.

**Character Sets and Collating Sequences**

The Guardian multibyte character-set procedures do not support all character sets supported by OSS. All Guardian multibyte character sets use two bytes per character. The Guardian multibyte character sets installed on a system can be determined by calling the MBCS_CODESETS_SUPPORTED_ procedure. These sets include:

- HP Kanji
- IBM Kanji
- IBM Kanji mixed
- JEF (Fujitsu) Kanji
- JEF (Fujitsu) Kanji mixed
- NEC Kanji
- JIS Kanji
- HP Hangul
- Chinese Big 5
- Chinese PC
- HP KS C 5601-1987
A default Guardian character set can be established, separate from that of the OSS locale in use; the Guardian default character set can be determined by calling the MBCS_DEFAULTCHARSET_ procedure.

Collating sequences for Guardian procedure calls are not affected by OSS locale or by the use of the Guardian multibyte character-set procedures. Individual procedure call features provide the only control over data collation.

**Date-Time Formats**

Date and time information returned or displayed by Guardian procedure calls are not affected by locale. Individual procedure call features provide the only format controls for both kinds of data.

**Message Text**

Run-time diagnostic messages do not vary by locale.

**Internationalization Functions and HP NonStop SQL/MX**

SQL/MX is not locale-sensitive and does not use the OSS internationalization features.

**Character Sets and Collating Sequences**

SQL/MX supports a subset of the character sets supported by OSS. For more information about SQL/MX character sets, see the *SQL/MX Reference Manual*. For information about using character sets in SQL/MX applications, see the *SQL/MX Programming Manual for Java* and the *SQL/MX Programming Manual for C and COBOL*.

Unlike SQL/MP, you cannot create collations for SQL/MX tables and cannot use SQL/MP collations on SQL/MX tables. SQL/MX Release 2.0 supports only the DEFAULT collation. The DEFAULT collation is based on binary ordering and is the default collating sequence for CHAR and NCHAR data types.

Each column of character data in an SQL/MX database is permanently associated with a character set when the column is created. You can specify a different character set for different columns in a table, but you cannot change the character set associated with a column after the column is created.

**Single-Byte Character Sets**

SQL/MX supports ISO88591. For more information, see the *SQL/MX Reference Manual*.

**Double-Byte Character Sets**

SQL/MX allows you to define a character column with UCS2 and to associate UCS2 with a string literal or host variable. If you are using SQL/MP tables with SQL/MX Release 2.0, you can also use KANJI and KSC5601 for string literals, host variables, or
SQL functions and predicates; however, you cannot use KANJI or KSC5601 character sets for character columns of SQL/MX tables.

When you install SQL/MX, you can set the national character set. The national character set is associated with NCHAR and NATIONAL CHARACTER data types and with N string literals. If you do not specify a national character set, the default is UCS2. For information about setting the national character set, see the *SQL/MX Installation and Management Guide*.

**Date-Time Formats**

SQL/MX accepts date-time values in three formats, which differ from OSS. SQL/MX includes a function that converts date-time values from one SQL/MX format to another. If you store date-time values in an SQL/MX or SQL/MP table, you must use the SQL/MX DATEFORMAT function to retrieve and format the values, not the OSS functions that vary by locale. SQL/MX does not recognize values from the OSS functions as date-time information. For more information about the DATEFORMAT function, see the *SQL/MX Reference Manual*.

**Message Text**

To receive message text in your embedded C, C++, or COBOL85 application in Unicode or ISO 8859-1 character set, you must declare the host variable with either of the two character sets UCS2 or ISO88591. SQL/MX does not allow KANJI or KSC5601 in this situation. Unlike SQL/MP, SQL/MX does not provide a DEFINE to specify an alternate message file that contains messages in another language.

**Internationalization Functions and HP NonStop SQL/MP**

SQL/MP is not locale-sensitive and does not use the OSS internationalization functions.

**Character Sets and Collating Sequences**

SQL/MP supports a subset of the character sets supported by OSS. For more information about SQL/MP character sets, see the *SQL/MP Reference Manual*. For information about using character sets in SQL/MP applications, see the *SQL/MP Programming Manual for C* and the *SQL/MP Programming Manual for COBOL85*.

**Single-byte Character Sets** SQL/MP supports nine single-byte character sets, ISO 8859-1 through ISO 8859-9. Users define collating sequences for a column. Each column can have a different collating sequence; a collating sequence is defined by the CREATE COLLATION command in the SQLCI utility.

Each column of character data in a SQL/MP database is permanently associated with a character set and a collating sequence when the column is created. If the column is part of the key for a table or index, the collating sequence for the column determines the storage order for rows within the table. You can specify a different character set and collating sequence for different columns in a table, but you cannot change the
character set or collating sequence associated with a column after the column is created.

You can specify a different collating sequence for a sort or comparison that involves single-byte character values within the same character set, but you cannot automatically vary that collating sequence based on locale.

**Double-byte Character Sets** SQL/MP supports two double-byte character sets—HP Kanji (KANJI) and HP Korean (KSC5601). Collation order and upshift rules for the SQL/MP double-byte character set differ from those provided in the OSS environment. Characters in the SQL/MP double-byte character sets always collate according to their binary values and cannot be upshifted.

You cannot specify a different collating sequence for double-byte character sets for a sort or comparison as you can with the SQL/MP single-byte character sets.

**Date-Time Formats**

SQL/MP accepts date-time values in three formats, which differ from those available to OSS functions. SQL/MP includes a function that converts date-time values from one SQL/MP format to another. If you store date-time values in an SQL/MP table, you must use the SQL/MP DATEFORMAT function to retrieve and format the values, not the OSS functions that vary by locale. SQL/MP does not recognize values from the OSS functions as date-time information. For more information about the DATEFORMAT function, see the *SQL/MP Reference Manual*.

**Message Text**

SQL/MP messages do not vary by locale. SQL/MP does, however, provide a DEFINE to specify an alternate message file that contains messages in a language appropriate for your application: `=_SQL_MSG_node.`

**Compiling Internationalized Applications**

The `c89` utility enables programmers to compile internationalized, ISO-compliant C programs. See the `c89(1)` reference page either online or in the *Open System Services Shell and Utilities Reference Manual* for details on the `c89` utility.

**Compiling in the TNS/R Native Environment**

In the OSS TNS/R native environment, `c89` provides an interface to these components of the C compilation system:

- The native C compiler
- A linking utility such as `nld` (for native object files that do not contain position-independent code) or `ld` (for position-independent code files)
- The SQL/MP compiler
- The SQL/MX preprocessor and compiler
To compile an internationalized application in the TNS/R native environment, run the native `c89` utility using this command:

```
c89 options myapp.c
```

**Compiling in the TNS Environment**

In the OSS TNS environment, `c89` provides an interface to these components of the C compilation system:

- C language preprocessor
- C compiler
- Binder
- Accelerator
- SQL/MP compiler

Before compiling an internationalized TNS application, use the VPROC command to ensure that the appropriate library is available:

```
VPROC $SYSTEM.SYSTEM.LI18N
```

If your TNS application requires the locales included in the T8372 libraries, you must use the nonnative `c89` utility with the `-l` library operand. The `-l` operand directs `c89` to use a specific library when binding your application. For example:

```
/nonnative/bin/ c89 options myapp.c -l i18n
```

In this example, `c89` compiles the application `myapp.c` and then dynamically binds the program, using the library defined in `libi18n.so`. If the appropriate locale library is not on the system when you use the `c89 -l` operand, the application defaults to the C/POSIX locale.

The `c89` utility performs dynamic linking using the TNS shared run-time library that includes the C/POSIX locale by default. If your application requires only the C/POSIX locale, you do not need to specify a library when using the `c89` utility.

**Testing Internationalized Applications**

Testing internationalized applications is a key part of ensuring that an application will survive in a global market. To test applications thoroughly, begin early in the product lifecycle. Review product specifications for internationalization and localization issues, then begin testing the internationalization and localization aspects of the product as early as possible.

Because the internationalization functions are part of an application’s core, solving problems early helps ensure that the overall product meets quality standards and that it is fully localizable.

This section offers testing guidelines and discusses how to verify that your application is successfully internationalized and is functioning properly for localization.
Basic Testing Checklist

Use this checklist to verify that the application complies with the basic code preparation requirements for internationalization:

- Is the program source code data transparent?
- Are all hard-coded messages removed from the source code and stored in a separate file or files?
- Is all culturally dependent information removed from the source code? Examples of culturally-dependent information include date, time, and monetary formats.
- Are all assumptions about character encoding removed from the source code and replaced with internationalized functions?

See Preparing Source Code on page 4-4 for a discussion of code preparation guidelines.

General Testing Guidelines

These guidelines can help ensure that the internationalization aspects of your application function properly:

- Verify that the test libraries are themselves internationalized. Use the setlocale(), nl_langinfo(), and localeconv() functions to return current locale information to your test libraries.

- Determine which locale categories will affect the behavior of the application you are testing, and concentrate tests on areas most likely to be affected. For example, if the isalpha() function is used, be sure that the only locale category affected is LC_CTYPE. You can determine the areas that affect the application’s behavior by finding the XPG4 internationalization functions used in the application, then checking that the categories are correctly affected. See Locale-Sensitive Functions on page 4-6 for a list of internationalization functions.

- Use the localized message catalogs to verify that the mechanism that accesses messages is functioning correctly and accesses messages appropriate to the locale setting. If localized message catalogs are not available for testing, you can create a dummy message catalog to verify that the internationalization mechanism works according to a selected locale. Be sure the dummy message source file contains the same number of sets and messages as the localized message catalog—the content of the file is not critical. Name the dummy message catalog according to the message catalog naming conventions to verify that the mechanism that accesses messages is functioning correctly.

- In addition to run-time testing of an application, conduct code inspections to review the source code.
Testing the Application’s Use of Locales

You can switch locale environment variables to verify that an application's behavior is consistent with the selected locale and that it accesses the appropriate message catalog. Test all supported locales to verify that the locale-sensitive aspects of the application are locale-independent, and that the proper set of internationalized functions has been used in development. Test each of the six locale categories—LC_CTYPE, LC_COLLATE, LC_MONETARY, LC_NUMERIC, LC_TIME, and LC_MESSAGES.

The following example shows the results of a test of the monetary format of an application. The application generates the value five thousand point five; if no locale-sensitive data is hard-coded, the output results in these monetary formats for the C/POSIX, US, and French locales:

<table>
<thead>
<tr>
<th>Locale</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/POSIX</td>
<td>5000.50</td>
</tr>
<tr>
<td>US</td>
<td>$5,000.50</td>
</tr>
<tr>
<td>French</td>
<td>5000,50F</td>
</tr>
</tbody>
</table>

The currency symbol for the US locale is $ and the monetary decimal point is .. When the locale is changed to the French locale, the currency symbol changes to F and the monetary decimal point to , . Note that this example shows only the formatting of a numeric value—it does not convert between currencies.

Troubleshooting Internationalized Applications

The key to troubleshooting problems in an internationalized application is determining whether the problem is in the user’s environment setting or in the application itself. This section offers tips to help in troubleshooting an internationalized application.

Problems With the User’s Environment

Users are responsible for providing the correct data in their environments. When troubleshooting problems in an internationalized application, ensure that the user’s locale setting for processing data is consistent with the locale in which it was created.

For example, if a user changes the locale from French to Spanish while using data that was created with the French locale, the application processes the data according to the rules for the Spanish locale. An internationalized application handles the data as a bit stream and encodes the data according to the current locale setting.

Identifying Problems in the Application

Determining whether the problem is in the application code itself or in its internationalization aspects begins with an understanding of how the components of an
internationalized application are interconnected. In internationalized applications the environment variables might affect the locales, which in turn affect character processing and the messaging system.

As Figure 4-1 on page 4-18 shows, the application inherits the internationalization environment variables. After the application sets the locale it calls the internationalization functions, which access the locale data.

These troubleshooting tips can help you locate the source of problems in an internationalized application:

- Did you follow the code preparation and testing guidelines discussed in Testing Internationalized Applications on page 4-15?
- Did you define the _IGNORE_LOCALE feature-test macro in your internationalized application? Program modules compiled with the _IGNORE_LOCALE feature-test macro defined use versions of the C library functions that do not support internationalization and locales. _IGNORE_LOCALE causes the application to use the C/POSIX default locale only.
- If operating in the TNS environment, did you use the -l library operand and specify the i18n library when compiling and binding your application with c89?
- Is setlocale() using the correct locale name?
- If starting the application from the shell? If you did, be sure that you exported the locale after setting it.
- Are the locale environment variables using valid variable precedence? Use the locale utility to verify variable precedence.
- Are the message source files correct?
• Are the code-set conversion function variables defined correctly? For example, if the `LOCPATH` environment variable is incorrectly defined the code-set converter or code-set conversion table object cannot be opened.
Software Supporting Multiple Character Sets

The HP products and commonly used third-party software listed in Table A-1 and Table A-2 on page A-3 have been evaluated by test labs local to the country where the character set is used. A Y indicates that the character set is supported while an N indicates that the character set is not supported.

A blank indicates that support has not been tested by a local test lab. Other character sets might be supported and other products also might support two-byte character sets.

Table A-1. Software Tested For Support of Unicode and Chinese Two-Byte Character Sets (page 1 of 2)

<table>
<thead>
<tr>
<th>Software</th>
<th>Unicode</th>
<th>Chinese</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UCS-2</td>
<td>UTF-16</td>
</tr>
<tr>
<td>HP Enterprise Toolkit -- NonStop Edition (ETK) version 1.n and later</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>NonStop Server for Java version 3.1 and later</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>NonStop JDBC server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop JDBC/MX version 2.0 and later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop Server for Java Message Service (JMS) version 2.0 and later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop Servlets for JavaServer Pages (JSP) version 2.0 and later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop SOAP version 2.4 and later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop ODBC/MX version 2.0 and later</td>
<td>Y *</td>
<td>Y **</td>
</tr>
<tr>
<td>OutsideView version 7.n and later</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>NonStop Pathway/TS</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>NonStop Remote Server Call/MP (RSC/MP)</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

* Correctly converted to target encoding.
** If there are no surrogate characters.
Table A-1. Software Tested For Support of Unicode and Chinese Two-Byte Character Sets  (page 2 of 2)

<table>
<thead>
<tr>
<th>Software</th>
<th>Unicode</th>
<th>Chinese</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UCS-2</td>
<td>UTF-16</td>
</tr>
<tr>
<td>SequeLink version 5.3 and later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spooler</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>NonStop SQL/MP</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>NonStop SQL/MX Release 1.(n) and later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop SQL/MX Release 2.0 and later</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>TCP/IP LAN print spooler</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>TN3270</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>NonStop TS/MP</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>NonStop TUXEDO</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Visual Inspect</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Extensible Markup Language (XML) parser</td>
<td></td>
<td></td>
</tr>
<tr>
<td>version 3.0 and later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop XSLT version 1.0 AAA and later</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Correctly converted to target encoding.
** If there are no surrogate characters.
<table>
<thead>
<tr>
<th>Software</th>
<th>Japanese</th>
<th>Korean</th>
<th>ISO 8859-1, when used by default</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Shift JIS</td>
<td>Kanji</td>
<td>KS C 5601-1987</td>
</tr>
<tr>
<td>HP Enterprise Toolkit -- NonStop Edition (ETK) version 1.1 and later</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop Server for Java version 3.1 and later</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>NonStop JDBC server</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>NonStop JDBC/MX version 2.0 and later</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>NonStop Server for Java Message Service (JMS) version 2.0 and later</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>NonStop Servlets for JavaServer Pages (JSP) version 2.0 and later</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>NonStop SOAP version 2.4 and later</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop ODBC/MX version 2.0 and later</td>
<td>*</td>
<td>*</td>
<td>Y</td>
</tr>
<tr>
<td>OutsideView version 7.1 and later</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>NonStop Pathway/TS</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop Remote Server Call/MP (RSC/MP)</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SequeLink version 5.3 and later</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spooler</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop SQL/MP</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>NonStop SQL/MX Release 1.1 and later</td>
<td>N</td>
<td>Y**</td>
<td>Y**</td>
</tr>
<tr>
<td>NonStop SQL/MX Release 2.0 and later</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>TCP/IP LAN print spooler</td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>TN3270</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

* Evenness of length is verified for this type of target, but SQL functions do not generate correct results.

** Treated as a single-byte data type
Table A-2. Software Tested For Support of Japanese and Korean Character Sets
(page 2 of 2)

<table>
<thead>
<tr>
<th>Software</th>
<th>Japanese</th>
<th>Korean</th>
<th>Shift JIS</th>
<th>Kanji (data type)</th>
<th>KS C 5601-1987</th>
<th>ISO 8859-1, when used by default</th>
</tr>
</thead>
<tbody>
<tr>
<td>NonStop TS/MP</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop TUXEDO</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visual Inspect</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extensible Markup Language (XML) parser</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>version 3.0 and later</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NonStop XSLT</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>version 1.0 AAA and later</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Evenness of length is verified for this type of target, but SQL functions do not generate correct results.

** Treated as a single-byte data type
**Glossary**

**ANSI.** The American National Standards Institute.

**Arabic-based writing system.** A writing system with letters that are derived from the Arabic alphabet. Not all languages that use Arabic characters are related linguistically to Arabic.

**ASCII.** The American Standard Code for Information Interchange. A single-byte code set that uses only 7 of the 8 bits in a byte to represent each character. The ASCII code set contains the uppercase and lowercase characters of the U.S. English alphabet, some punctuation symbols, the digits 0 through 9, and some symbols and control characters. Because of its limited characters, and because the 8th bit is sometimes used in ASCII programs as a utility bit, the ASCII code set is not appropriate for use in international software.

**base character.** A character that can be combined with one or more combining characters to form a composite character.

**Basic Multilingual Plane (BMP).** The lower two octets, row and cell, of the ISO 10646 character layout. Also known as Universal Coded Character Set - 2 (UCS-2).

**block-based writing system.** A writing system composed of single letters that stand alone in printed text such as English, French, and Russian.

**BMP.** See Basic Multilingual Plane (BMP).

**byte.** An ordered set of bits that represents a character or a part of a character. The number of bits per byte is implementation-dependent; a byte usually contains 8 or more bits. Also called an octet.

**C locale.** A special locale defined by the ANSI C standard. Every standard C program always starts up in the C locale, which means that no locale-specific action takes place, and the program operates in the ASCII mode. All library functions behave as they do in standard C. Unless the program calls the setlocale() function, none of the behavior changes. Also called the POSIX locale or the C/POSIX locale.

**character.** A sequence of one or more bytes representing a single character; used for the organization, representation, or control of data. A single-byte character consists of eight bits that represent a character. A multibyte character uses one or more bytes to represent a character. A wide character is a fixed-width character wide enough to hold any coded character supported by an implementation.

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

**character encoding.** A method in which each member of a character set is mapped to specific numeric code values.
Character set. A finite set of characters (letters, digits, symbols, ideographs, or control functions) used for the organization, representation, or control of data. See also code set.


code set. Codes that map a unique numeric value to each character in a character set, using a designated number of bits to represent each character. Single-byte code sets use 7 or 8 bits to represent each character. The ASCII and ISO 646 code sets use 7 bits to represent each character in Roman-based alphabets; these code sets are very limited and are not appropriate for international use. The single-byte ISO 8859 code sets use 8 bits to represent each character and can therefore support Roman-based alphabets and many others including Greek, Arabic, Hebrew, and Turkish. Multibyte code sets represent characters that require more than one byte, such as East Asian ideographic characters.

collation. The logical ordering of characters or character strings according to defined precedence rules. The precedence rules identify a collation sequence between the collating elements and additional rules that can be used to order strings consisting of multiple collating elements.

combining character. One or more characters that can be combined with a base character to form a composite character.

Common Run-time Environment (CRE). An HP product that provides a common environment for applications written in different programming languages.

composite character. A character consisting of a combination of two or more elements in a single character position. For example, the base character 'a' and the acute accent (' ′) diacritical mark can be combined to form the composite character 'á'. The ISO 10646 code set includes the encoding method enabling creation of composite characters.

context-dependent writing system. A writing system in which characters take different forms depending on their location within a word—for example, Arabic and Hebrew.

cultural data. Information that can vary from one language to another or between geographic areas. Date, time, and currency formats are examples of cultural data.

data transparent. Describes software that examines all eight bits of every data byte and that uses no bit in a data byte for its own purposes. Internationalized applications must be data-transparent.

diacritical. A mark added to a letter that usually provides information about how the letter should be pronounced or the stress that should be given to a syllable. Examples include the acute accent (' ′), the grave accent (' ″), the diareses (' ″), and the tilde (~).

eight-bit clean. Software that processes eight-bit characters without modifying or using any bit in a data byte for its own purposes. Also known as data transparent or 8-bit transparent.
encoding method. A set of rules for combining two or more code sets into a single data stream.

environment variable. A variable that is associated with a specific area of a user’s environment. For example, the LC_TIME locale environment variable enables the display of time formats according to local time conventions.

EUC. The Extended Unix Codes. EUC is an encoding method most commonly used on Asian UNIX-based systems. Up to four code sets can be combined in a single data stream using EUC.

file code. A bit pattern representing a character in a character set. A file code is an external representation of data and is the format of data stored on disk.

function. A service called by an application program to communicate with other software components. For example, the setlocale() function establishes the locale in which an internationalized program is to work.

GB. Guo Biao, which means “national standard” in Chinese. This term is commonly used in standards published in the People’s Republic of China.

gencat. An XPG standard command that generates message catalogs from message source files.

genxlt. A utility that generates code set conversion table objects from code set conversion table source files.

Guardian. An environment available for interactive or programmatic use with the HP NonStop Kernel operating system. Processes that run in the Guardian environment usually use the Guardian system procedure calls as their application program interface; interactive users of the Guardian environment usually use the HP Tandem Advanced Command Language (TACL) or another HP product’s command interpreter. Contrast with Open System Services (OSS).


Hangul. The Korean phonetic writing system.

Hanja. The Korean ideographic writing system.

Hanzi. The Traditional and Simplified Chinese ideographic writing systems.

Hiragana. One of two Japanese phonetic character sets (the other is Katakana). Hiragana characters are mixed with Kanji characters to write all of or parts of Japanese words phonetically.

iconv. An XPG standard utility that converts encoded characters from one code set to another.
ideograph. A character or symbol representing a word or idea. Some writing systems, such as Japanese and Chinese, use thousands of ideographs.

IEEE. Institute of Electrical and Electronics Engineers. IEEE is a professional organization whose committees develop and propose computer standards that define the physical and data link protocols of entities such as communication networks. IEEE formed the POSIX standard.

internationalization. The process of designing and coding software so it can be adapted to meet the needs of different languages, cultures and coded character sets, with the ability to handle a variety of linguistic and cultural conventions. Also referred to as "I18N", which is an abbreviation derived from the 18 letters between the between the initial “I” and final “N” of the word “internationalization.”

ISO. International Organization for Standardization. ISO is an international body that drafts, discusses, proposes, and specifies standards for network protocols. ISO is best known for its seven-layer reference model that describes the conceptual organization of protocols.

ISO is sometimes called the "International Standards Organization"; although ISO is the official abbreviation, it does not correspond to the organization’s name in any language.

ISO 10646. A universal coded character set (UCS) established by ISO to represent all characters and symbols from all commonly used languages. ISO 10646 characters are encoded in multiple octets in which code space is divided into four units—group, plane, row, and cell. Includes a combining character method of encoding to allow any number of combining characters to follow a base character.

ISO 8859. A series of ISO standard 8-bit code sets used to represent languages based on many alphabets, including Roman, Greek, Cyrillic, Hebrew, Turkish, and Arabic. The ISO 8859 code sets are used in international applications that must be data transparent. ASCII is a subset of each of the ISO 8859 code sets.

JIS. The Japanese Industrial Standard. JIS is a group that creates standard code sets for Japanese.

Kana. A collective name for Katakana and Hiragana (the two types of phonetic Japanese characters).

Kanji. An ancient ideographic form of writing used in Japan; the form originated in China.

Katakana. One of the two Japanese phonetic character sets (the other is Hiragana). Japanese keyboards are often labeled with Katakana to enable users to type Japanese words phonetically.

KS. The Korean Standard. KS is a group that creates standard code sets for Korean.
**language.** System of communication made up of words formed by combinations of patterns and symbols and can vary depending on the people of a particular country or by groups with a shared set of history or tradition.

**Latin-based writing system.** A writing system with letters that are derived from the Latin alphabet. Not all languages that use Latin characters are related linguistically to Latin.

**locale.** (1) The subset of a user's environment that depends on language and cultural conventions. It is made up of or more categories, each of which controls specific aspects of the behavior of components of the system. Locale refers to a collection of rules and text that represent linguistic and cultural conventions for a specific region. (2) A standard utility that displays information about locales.

**locale source file.** A file containing linguistic and cultural source definitions for a locale.

**localization.** The process of adapting computer interfaces, data, and documentation to the culturally accepted way of presenting information. Sometimes referred to as "L10N," derived from the 10 letters between the initial "L" and the final "N" of the word "localization."

**message catalog.** A machine-readable file containing program messages, command prompts, and responses to prompts for a particular language, territory, and code set.

**message source files.** A file containing program messages that are processed by the gencat utility to produce a message catalog.

**messaging system.** A name for the process that involves separation of program messages from program code, the generation of a message catalog, and access to messages from the catalog.

**mkcatdefs.** A utility that preprocesses a message source file.

**multibyte character.** A coded character that uses one or more bytes to represent data. Multibyte data streams can include characters with varying widths.

**octet.** An ordered set of bits that represents a character or a part of a character. The number of bits per octet is implementation-dependent; the number of bits in an octet is usually 8. Also called a byte.

**Open System Services (OSS).** An open system environment available for interactive or programmatic use with the HP NonStop Kernel operating system. Processes that run in the OSS environment usually use the OSS application program interface; interactive users of the OSS environment usually use the OSS shell for their command interpreter. Synonymous with "Open System Services (OSS) environment." Contrast with Guardian.

**OSS.** *See Open System Services (OSS).*
PC. Personal Computer codes. PC codes are an encoding standard that is popular on East Asian personal computers.

POSIX. The Portable Operating System Interface, as defined by the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI). Each POSIX interface is separately defined in a numbered ANSI/IEEE standard or draft standard. The application program interface (API), known as POSIX.1, has become ISO/IEC IS 9945-1:1990.

POSIX locale. A special locale defined by the POSIX standard. Every standard C program always starts up in the POSIX locale, which means that no locale-specific action takes place, and the program operates in the ASCII mode. All library functions behave as they do in standard C. Unless the program calls the `setlocale()` function, none of the behavior changes. Also called the C locale or the C/POSIX locale.

process code. A uniform internal representation of a character from a character set. A process code is the format in which programs manipulate data.

Shift-JIS. The name commonly used for the microcomputer encoding scheme for the JIS standard. Shift-JIS implements JIS in 8-bit format, allowing direct (unescaped) access to the 128 ASCII characters for compatibility.

TNS. HP computers that support the HP NonStop Kernel and that are based on complex instruction-set computing (CISC) technology. TNS processors implement the TNS instruction set.

TNS/R. HP computers that support the HP NonStop Kernel and that are based on reduced instruction-set computing (RISC) technology. TNS/R processors implement the RISC instruction set and are upwardly compatible with the TNS system-level architecture.

UCS-2. The Universal Coded Character Set-2. The lower two octets, row and cell, of the ISO 10646 character layout. Also known as the Basic Multilingual Plane (BMP).

UCS-4. The Universal Coded Character Set-4. All four octets, group, plane, row and cell, of the ISO 10646 character layout.

Unicode. Code set that currently uses the two lower octets, row and cell, and does not support levels of combining characters. Equivalent to ISO 10646 UCS-2, Level 3.

Uniforum. An international association of open-systems professionals that contributes to various standards.

wide character. A fixed-width character that is wide enough to hold any coded character your implementation supports. A wide character is an object of the `wchar_t` type definition.

XPG. An acronym that stands for the X/Open Portability Guide.

XPG4. Commands and utilities that comprise a superset of the POSIX utilities.
Index

A
Accent marks 2-1
Algorithmic code-set converters 3-13, 4-2
Applications using Guardian procedure calls 4-11
Arabic-based writing systems 2-2
ASCII 2-5

B
Base characters 2-7
Basic Multilingual Plane (BMP) 2-7
Big 5 2-9
Big 5 character set A-1
Block-based writing systems 2-1

C
c89 utility 4-1, 4-14
catclose() 3-10
catgets() 3-10
catopen() 3-10
Character 2-13
classification 2-13
combining 2-7
composite 2-7
data types 2-9, 2-11
encoding 4-6
mapping 2-16
multibyte 2-10
sets 2-4
single-byte 2-10
wide 2-10
Chinese 2-3, 2-6
Chinese National Standard (CNS) 2-6
COBOL applications 4-10

Code preparation
character encoding 4-6
culturally dependent information 4-5
data transparency 4-4, 4-5
for internationalization 4-4
isolating messages from source code 4-5

Code sets 2-5
ASCII 2-5
Big 5 2-9
Chinese National Standard (CNS) 2-6
combining characters 2-7
composite characters 2-7
East Asian 2-6
encoding methods 2-9
Extended UNIX Codes (EUC) 2-9
Guo Biao (GB) 2-6
ISO 10646 2-7
ISO 8859 2-5
Japanese Industrial Standard (JIS) 2-6
Korean Standard (KS) 2-6
multibyte 2-6
personal computer codes 2-9
shift-JIS 2-9
single-byte 2-5
Unicode 2-8
Universal Coded Character Set (UCS) 2-7

Code-set conversion 3-11
accessing conversion tables 3-13
algorithmic converters 3-13
conversion-table source files 3-12
converters for the TNS environment 4-4
Index

Code-set conversion (continued)
converters for the TNS/R native environment 4-2
generating conversion tables 3-12	
table-driven converters 4-3

Collation
character-encoded 2-14
character-set 2-15
don’t-care characters 2-16
ideographic character 2-15
multilevel 2-15

Combining characters 2-7

Compiling internationalized applications
in the TNS environment 4-15
in the TNS/R native environment 4-14

Composite characters 2-7

Context-dependent writing systems 2-2

Conversion, code-set 3-11
accessing conversion tables 3-13
algorithmic converters 3-13, 4-2
conversion-table source files 3-12
converters for the TNS environment 4-4
converters for the TNS/R native environment 4-2
generating conversion tables 3-12	
table-driven converters 4-3

Culturally dependent information 3-6, 4-5

Currency symbol 2-18

Currency symbols 2-18

Cursive writing systems 2-2

C/POSIX locale 3-4, 3-5, 3-6, 4-1, 4-9

Data types (continued)
single-byte characters 2-10
unsigned characters 2-11
wide characters 2-10

Data types
character 2-9, 2-11
integer 2-11
multibyte characters 2-10
signed characters 2-11

Date formats 2-16

Default locale (C/POSIX) 3-4, 3-5, 3-6, 4-1, 4-2, 4-9

Development guidelines for internationalized applications 4-4

Diacritical marks 2-1
dspcat utility 3-8
dspmsg utility 3-8

E

East Asian code sets 2-6
East Asian writing systems 2-2
Eight-bit clean 2-12, 4-5
Eight-bit transparency 2-12, 4-5
Encoding methods 2-9
Extended UNIX Codes (EUC) 2-9

F

File codes 2-10, 2-11
FILE_OPEN_ 4-9

G

GB 18030 character set 2-6
GB 2312 character set 2-6, A-1
gencat utility 3-7, 3-9
genxlt utility 3-12
GeoReady 1-1, 4-1
Guidelines for internationalized applications
design and development 4-4
testing 4-15
troubleshooting 4-17

Guo Biao (GB) 2-6
Index

H
Han 2-3
Hangul 2-4
Hanja 2-3
Hanzi 2-3
Hiragana 2-4
HP internationalization subsystem 1-5

I
iconv() 3-13
iconv_close() 3-13
iconv_open() 3-13
Ideographic character collation 2-15
Ideographs 2-2
IEEE 1-5
Integer data types 2-11
International Organization for Standardization (ISO) 3-1
Internationalization 1-1
and character sets 2-4
and code sets 1-2, 2-5
and culture-specific data 1-2, 1-4
and language-specific data 1-1, 1-2, 1-4
definition 1-1
development considerations 4-4
functions 3-6
how to begin 1-4
HP subsystem 1-5
locale-sensitive functions 4-6
POSIX and XPG model 3-2
standards 1-5, 3-1
Tandem subsystem 3-2, 3-3, 4-1/4-19
testing internationalized applications 4-15
troubleshooting problems 4-17
utilities
dspcat 3-8
dspmsg 3-8
gencat 3-7, 3-8, 3-9

Internationalization (continued)
utilities (continued)
genxlt 3-12
mkcatdefs 3-7, 3-9
runcat 3-8, 3-9
ISO 3-1
ISO 8859-1 character set 2-5, A-3

J
Japanese 2-4
Japanese Industrial Standard (JIS) 2-6

K
Kana 2-4
Kanji 2-3, 2-4
Kanji character set 2-3, 2-4, A-3
Katakana 2-4
Korean 2-4
Korean Standard (KS) 2-6
KS C 5601-1987 character set 2-7, A-3

L
LANG 3-4, 3-5
Languages 2-1
Latin-based writing systems 2-1
LC_ALL 3-4, 3-6
LC_COLLATE 3-4, 4-17
LCCTYPE 3-4, 4-17
LC_MESSAGES 3-4, 4-17
LC_MONETARY 3-4, 4-17
LC_NUMERIC 3-4, 4-17
LC_TIME 3-4, 4-17
Locale
and case conversion 1-4
and character classification 1-4
and collating sequences 1-4
default (C/POSIX) 3-4, 3-5, 3-6, 4-1, 4-9
definition 1-4, 3-2
Locale (continued)
in client/server applications 4-9
locale-sensitive functions 4-6
multibyte 4-1, 4-2
name 3-3
objects 3-3
single-byte 4-1, 4-2
supported locales 4-1, 4-2
testing 4-17
Locale environment variables
LANG 3-4
LC_ALL 3-4
LC_COLLATE 3-4
LC_CTYPE 3-4
LC_MESSAGES 3-4
LC_MONETARY 3-4
LC_NUMERIC 3-4
LC_TIME 3-4
precedence rules 3-4
program environment 3-6
user environment 3-5
Locale-independent code 3-6
Locale-sensitive functions 4-6
Localization 1-3
LOCPATH 3-13

Message catalog
accessing 3-10
generating 3-9
Message source file, creating 3-8
Messaging system
accessing a message catalog 3-10
creating a message source file 3-8
definition 3-7
dspcat utility 3-8
dspmsg utility 3-8
gencat utility 3-7, 3-8
generating a message catalog 3-9

Messaging system (continued)
isolating messages from source code 4-5
mkcatdefs utility 3-7, 3-9
runcat utility 3-8, 3-9
mkcatdefs utility 3-7, 3-9
Monetary formats 2-18
Multibyte
characters 2-10, 2-11
code sets 2-6
locales 4-1, 4-2
Multilevel collation 2-15

NLSPATH 3-10, 3-11
Numeric representation
date formats 2-16
monetary formats 2-18
time formats 2-17

Octet 2-7

PC 2-9
Personal computer (PC) codes 2-9
POSIX 1-5, 3-1, 4-1
POSIX internationalization model 3-2
POSIX standards 4-4
Precedence rules, locale environment variable 3-4
Preparing source code 4-4
Problems with internationalized applications 4-17
Process codes 2-11, 2-12
Program locale environment 3-6
Radix character 2-18
runcat utility 3-7, 3-9

setlocale() 3-6
setlocale_from_msg() 4-9
Shift JIS character set 2-9, A-3
Shift-JIS 2-9, 3-13
Signed character data types 2-11
Simplified Chinese 2-3
Single-byte
  characters 2-10
  code sets 2-5
  locales 4-1, 4-2
SJIS 2-9, 3-13
SJIS_AJEC 3-13
Source code
  data transparency 4-4
  isolating messages from 4-5
  preparing for internationalization 4-4
  removing character encoding assumptions from 4-6
  removing culturally dependent information from 4-5
SQL/MP applications 4-13
SQL/MX applications 4-12
Standards
  International Organization for Standardization (ISO) 3-1
  internationalization 1-5, 3-1
  organizations 3-1
  POSIX 3-1
  POSIX and XPG4 4-4
  Uniforum 3-1
  XPG 3-1

Table-driven code-set converters 4-3

Tandem internationalization subsystem 1-5, 3-2, 3-3, 4-1/4-19
Testing internationalized applications 4-15
The 4-2
Thousands separator 2-18
Time formats 2-17
TNS environment
  code-set converters 4-4
  compiling internationalized applications in 4-15
TNS/R native environment
  code-set converters 4-2
  compiling internationalized applications in 4-14
  table-driven converters 4-3
Traditional Chinese 2-3
Translation 1-3
Troubleshooting internationalized applications 4-17

UCS 2-7
UCS-2 character set 2-8, A-1
Unicode 2-8
Uniforum 3-1
Universal Coded Character Set (UCS) 2-7
Unsigned character data types 2-11
User locale environment 3-5
UTF-16 character set A-1
Utilities, internationalization
  gencat 3-7, 3-9
  genxlt 3-12
  mkcatdefs 3-7, 3-9
  runcat 3-9

Variables, locale environment 3-3
W
wchar_t 2-10
Wide characters 2-10, 2-11, 2-12
Writing systems 2-1
  Arabic-based 2-2
  context-dependent 2-2
  cursive 2-2
  East Asian 2-2
  ideographic 2-2
  Latin-based 2-1

X
READUPDATE 4-9
XPG 1-5, 3-1
XPG internationalization model 3-2
XPG4 4-1
XPG4 standards 4-4
X/Open 1-5, 3-1

Special Characters
$RECEIVE 4-9