FUJITSU COMPUTERS SIEMENS

SINIX/windows ONLINE Documentation

SINIX V5.41
Programer's Guide: Internationalization - Localization

Edition March 1993

---

1 Preface

Nowadays software products are used in networked systems which cross national boundaries. The product users speak different languages and conform to different local conventions and customs. Thus it is essential to be able to develop application software which can communicate with users in a variety of languages and make allowance for the cultural conventions of the countries or regions the users come from.
In the past the usual solution offered to customers with a requirement for national variants of a program was to modify and where necessary replace existing information based on language and cultural conventions directly in the source code. However, the consequence of this approach is that national variants of a program generate additional costs owing to the extra processing effort, take longer to reach the market, and may spawn numerous different versions of a program, which in turn makes version management more time-consuming.
Like fellow members of the X/Open group, Siemens Nixdorf Informationssysteme AG adopts a different approach to supplying customers with national program variants. There is now no need to write and compile a series of different national variants: in the SINIX V5.41 operating system, all information dependent on a particular language or country has been removed from the source files. All essential information about the language of the users and any data specific to a particular country or language are held separate from the program logic. This process is known as *internationalization*.
When an internationalized application program executes, its runtime environment is dynamically supplied with the native language which has been set and the corresponding national or regional conventions. The process of establishing information specific to a language variant within a computer system is known as *localization*. The set of conventions specific to a particular country or language is known as the *locale*.
If a program variant is required for another language or country, it is now longer necessary to modify the sources; only the data specific to the language or country needs to adapted and where appropriate integrated in a new locale.

Note:

Some of the functionality described in this guide may not be supported by the hardware you have or by your release of the SINIX V5.41 operating system. Refer to the operating system release notes to find out how much of the functionality is available to you.

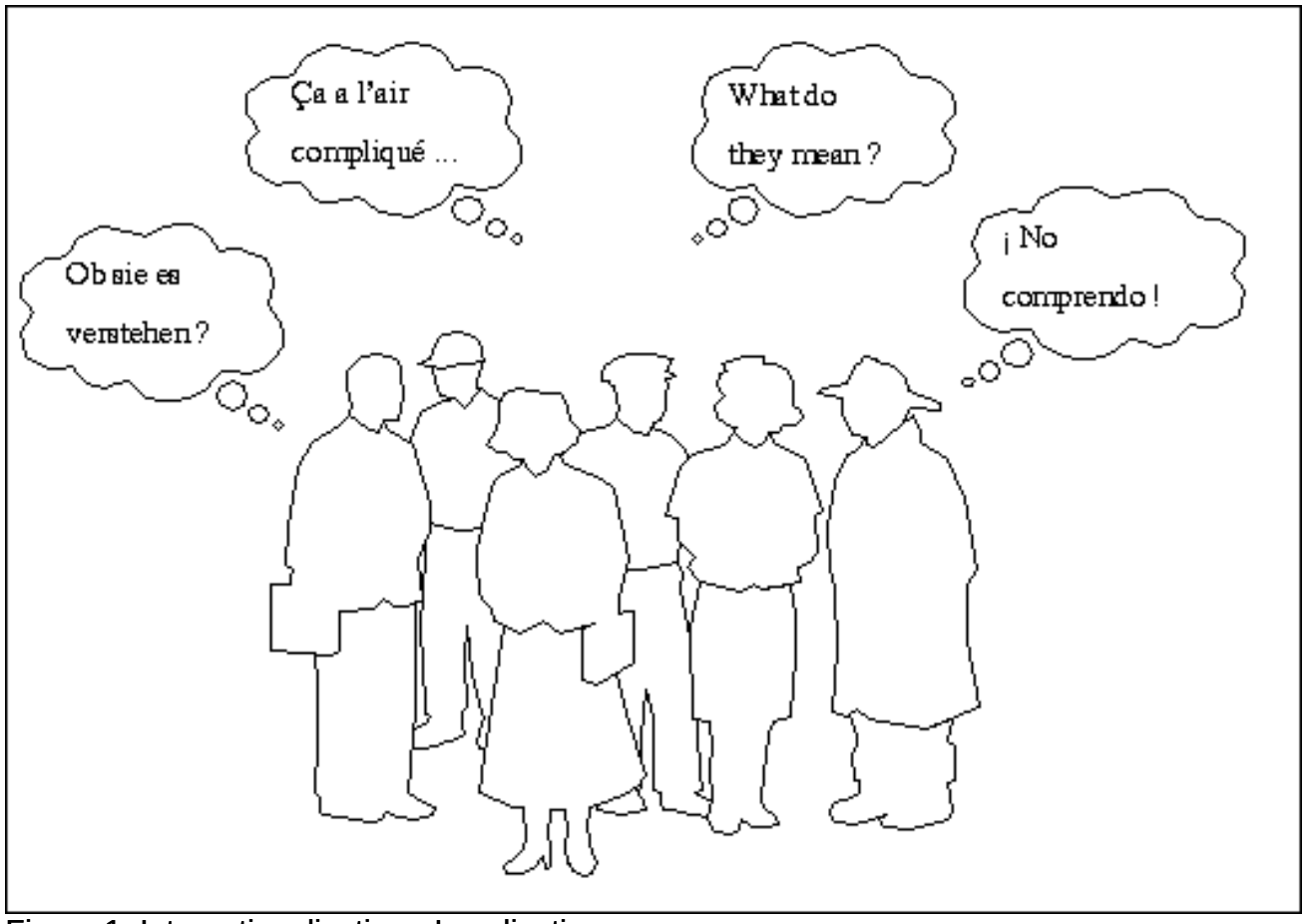

Figure 1: Internationalization - Localization

1.1 Target group

This manual describes how to use the functions and commands available as part of the SINIX V5.41 operating system to internationalize software products and how to localize internationalized software products on the system.
It is in the form of a guide, and it does not aim to provide comprehensive specifications of the individual interfaces. Detailed information on these interfaces is given in the manuals referenced in the appropriate chapters of this guide.
The guide is intended for product planners and developers of new internationalized products and for those needing to internationalize and enhance existing products. It is also intended for system administrators and users of SINIX systems who want to make use of the support for multilingual operation which internationalized software provides.

1.2 Summary of contents

This manual concentrates on two main topics:

- developing internationalized software
- localizing internationalized software

Chapter 2, "Aspects of internationalization", is meant for the entire target group and provides an introduction to the problems involved in running software products in a multilingual environment. It aims to give you some insight into the many aspects of internationalization and to make you aware of the differing needs of product users.

The central chapters 3, 4 and 5 can be read independently of each other. Chapters 3 and 4 are intended primarily for product planners and developers; chapter 5 is primarily for system administrators and users of internationalized software.

Chapter 3, "Developing international software", describes the process of developing internationalized software and also how to internationalize existing software. Chapter 4, "Localizing internationalized software", deals with generating locale-specific databases, which are accessed by internationalized application programs at runtime.

Chapter 5, "Using internationalized software", discusses the needs of internationalized software users. It examines the requirements for working with internationalized software, i.e. settings for system-wide and user-defined environments offering the chosen languages and country-specific conventions.

1.3 Related documentation

If you want to explore the subject of internationalization further and need in-depth information about the functions and commands dealt with in this guide, the manuals discussed below will be useful.
For information on ordering these publications refer to the References at the back of this manual.

X/Open Internationalisation Guide "[14]"
The X/Open Internationalisation Guide describes the internationalization functions of systems based on the X/Open Common Applications Environment. It is intended for software developers writing internationalized software and for users needing to utilize the multilingual operation functionality of X/Open-compliant systems.
As the SINIX V5.41 operating system is also XPG4-compliant, the information in the X/Open Internationalisation Guide is also applicable to internationalization for SINIX V5.41.

IHB Internationalization Handboo "[8]"
The IHB Internationalization Handbook issued by Siemens Nixdorf Informationssysteme AG provides a basic system-neutral introduction to the central problems of internationalization. It is a reference work defining the basic structure of an internationalized product and supplying rules, recommendations and helpful notes for creating international products. There are two companion manuals: the **IHB Checklist** "[9]", which is a catalog of questions for checking compliance with internationalization requirements, and the **IHB Tables Supplement** "[10]", which provides an overview of country-specific data.

C-DS C V1.0 Programmer's Reference Manual "[6]"
The C-DS C V1.0 Programmer's Reference Manual is intended for anyone working with the C Development System and writing C programs. It is a reference work describing C program development commands, system calls, libraries and functions plus C-specific file formats. The manual includes detailed information on all the commands, libraries and functions needed for internationalization.

SINIX V5.41 Commands, Volumes 1-3 "[1]" "[2]" "[3]"
Volumes 1-3 of the Commands manual for SINIX V5.41 contain descriptions of the SINIX user commands. They are a fundamental work of reference for all users of SINIX and provide detailed information about all the important commands.

1.4 Notational conventions

This manual uses the following notational conventions:

- Inner running title: the inner running title gives the name of the current chapter.
- Outer running title: the outer running title gives the name of the current section.

Input

- All input lines have to be terminated with the return key **[RETURN]**. Consequently, the return key **[RETURN]** is not explicitly depicted at the end of such lines.
- In continuous text, input is shown in `fixed-width font`.

Output

- Operating system output appears in `fixed-width font`.

Command syntax

Where command descriptions go into detail, they are presented in the following format:

- The standard syntax of a command is shown in `fixed-width font`. Variable parts of a command line are shown in *italics*. Variables must be replaced with suitable values (such as file names); any characters which are not in italics must be entered exactly as shown.
- Presentation of options: options modify the way in which a command functions. They generally consist of a letter preceded by a minus sign *(-)*.
  The following syntax rules apply to options:
  - You can enter options in any order.
  - There are two ways of entering options which take no arguments:
    separately: *command* `-a -b -c`
    grouped together: *command* `-abc`
  - Options requiring an argument, e.g. `-d`*argument* or `-f` *argument*, cannot be grouped with arguments which do not take an argument. Options which take an argument are entered separately with blanks to separate them:
    command `-abc -d`*argument* `-f` *argument*
  - If only one of two options is to be entered, the two options are shown with the OR symbol | between them.
- Optional input, i.e. input which can be omitted without preventing the command from being executed, is shown within square brackets [ ]. The square brackets themselves must not be entered.

References

The following examples illustrate the forms in which reference is made to this manual, other manuals and any other reference material.

Other sections and chapters

see the chapter "Preface"	Reference to the chapter "Preface" of this manual.
see the section "Multibyte characters"	Reference to the section entitled "Multibyte characters" of this manual.

Other SINIX manuals

refer to the "System	Reference to the manual listed as

Administrator's Reference Manual" [4]	number [4] in the References at the back of the manual.

Other publications

refer to the "X/Open Guide, XPG3-XPG4 Base Migration Guide" [11]	Reference to a publication listed as number [11] in the References at the back of the manual.

Examples

The heading **Example:** is used to introduce examples.
Please note that the data examples are based on is not necessarily included in your system. If you want to try them out for yourself, you may first have to create the files, directories and so on that they require. Detailed instructions are given in the text.

Notes

The heading **Note** is used to highlight important information.

2 Aspects of internationalization

The SINIX operating system, like most UNIX derivatives, always used to be based on the ASCII coded character set, with American English as the language in which the user communicated with the computer. For the commercial market, however, it has always been essential to offer interactive programs communicating in the language of the program user (language in this sense being taken to include regional conventions such as currency formats); and as a result considerable extra effort was required to produce national variants of programs for other markets.
The growing international popularity of UNIX has now made it necessary to enhance the system to cater flexibly for the differing scripts, languages and cultural conventions of its users.

2.1 Character sets

Software products are used in many different countries, as well as in countries in which a number of different languages are spoken. Thus they need to be capable of handling a variety of languages and the associated characters such as letters, numbers and special symbols, for example in the Latin scripts which are common throughout Europe.
Characters used in one or more languages are combined to form a character set and made available by the computer's operating system. Within the computer these characters are represented in the form of code tables (see "IHB - Internationalization Handbook" [8] and "X/Open Guide, System Interface Definitions" [13]).
A coded character set, or codeset, consists of a defined group of letters, digits and symbols, plus ideograms in ideographic scripts (for some Asiatic languages), which is used to represent and organize data. Internally each of these characters occupies a fixed position in a code table. A code table may also include control characters and undefined positions.
In the past most UNIX systems were based on the ASCII (**A**merican **S**tandard **C**ode of **I**nformation **I**nterchange) 7-bit character set. The ASCII code table defines the 7-bit codes which represent the characters within a computer. These 7-bit codes among other things allow all 26 letters of the English language to be represented in uppercase and lowercase; the ASCII code table defines codes for a total of 128 characters.
There is a reproduction of the ASCII code table (ISO 646-IRV) in the section "Appendix B: Code tables". For an extensive selection of code tables refer to the manual "IHB Tables/IHB Tabellen" [10].
Character sets for other languages contain more characters than are available in the ASCII code table. Language-specific characters such as the German mutated vowels (umlauts) ä, ö and ü are not in the ASCII code table. For that reason the ISO (**I**nternational **O**rganisation for **S**tandardisation) initially set aside 12 special characters in the 7-bit character set as nationally assignable, thereby allowing various accented letters and umlauts to be represented. The downside of this, however, was double assignment of code positions for alphanumeric characters and special characters.
UNIX/SINIX systems today use a collection of ISO character sets based on ISO 8859 Standard for the major European languages. These 8-bit characters sets avoid the double assignment problem. Some of these character sets meet the character requirements of a number of languages in a single set.

The characters are represented internally by an 8-bit code, with the first 128 codes being the same as the ASCII code table and the other 128 being mapped to other printing and control characters. As the ASCII character set is included in all ISO 8859 code tables, languages other than English can be supported without any negative impact on existing non-internationalized products.
There is a reproduction of an ISO code table (ISO 8859-1) in the section "Appendix B: Code tables". For an extensive selection of code tables refer to the manual "IHB Tables/IHB Tabellen" [10].
To cater for local requirements, it must be possible for users to access a number of different languages in international products. Thus the neutral product core must make no assumptions about which character set is being used (refer to "IHB - Internationalization Handbook" [8]).

Note:

The minimum requirement for an internationalized product is 8-bit coding.

The following table illustrates the language-related areas of use for ISO 8859 character sets.

ISO standard	Name	Area of use	Languages
ISO 8859-1	ISO Latin-1	Western Europe	Danish, Dutch, Faroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish, English
ISO 8859-2	ISO Latin-2	Eastern and Central Europe	Albanian, Czech, German, Hungarian, Polish, Romanian, Serbo-Croat, Slovakian, Slovenian, English
ISO 8859-3	ISO Latin-3	Southern Europe	Afrikaans, Catalan, Dutch, Esperanto, French, German, Italian, Maltese, Spanish, Turkish, English
ISO 8859-4	ISO Latin-4	Northern Europe	Danish, Estonian, Finnish, French, German, Greenlandic, Lapp, Latvian, Lithuanian, Norwegian, Swedish, English

ISO 8859-5	ISO Latin-Cyrillic	Eastern Europe	Cyrillic characters, English
ISO 8859-6	ISO Latin-Arabic	Middle East (Arabic)	Arabic characters, English
ISO 8859-7	ISO Latin-Greek	Greece	Greek characters, English
ISO 8859-8	ISO Latin-Hebrew	Middle East (Hebrew)	Hebrew characters, English
ISO 8859-9	ISO Latin-5	Western Europe, Turkey	Variant of ISO Latin-1 (with Turkish but without Icelandic and Faroese)

Table 4: Areas of use for ISO 8859 character sets

2.1.1 8-bit transparency

8-bit transparency refers to the ability of a software product or component to process 8-bit characters without modifying or utilizing any part of the character in a way that is inconsistent with the rules of the current coded character set.
In the ASCII character set every character is encoded as a unique 7-bit sequence but is stored as an 8-bit unit. In the past the 8th bit was frequently used for program-dependent purposes. 7-bit coding restricts the maximum number of characters to 128, control characters included, so there was little scope for processing and representing characters from 8-bit character sets, which can handle a larger range of characters, so as to support other native languages and national conventions.
In 7-bit ASCII character handling, the 8th bit of a byte, known as the most significant bit or MSB, is used for validation or for flagging internal program states. Thus when converting programs to 8-bit it is also necessary to check whether there are any logical bit operations or masking operations which modify data bytes. For further information see section "Character classification (LC_CTYPE category)", section "Character conversion (LC_CTYPE category)" and section " Internationalizing an existing program".
Programs intended to process characters from 8-bit character sets must treat all 8 bits of a byte as part of the coding for the character itself. Programs which meet this condition are said to be *8-bit transparent* or *8-bit clean*. Programs capable of handling characters with more than 8 bits are said to provide *multibyte support* (see "Multibyte characters"). Such programs are *n-bit transparent*, where *n* is greater than 8 and is typically 8 or 16.
All the SINIX commands described in the manuals "SINIX V5.41 Commands, Volumes 1-3″ [1-3] and "SINIX V5.41 System Administrator's Reference Manual" [4] are 8-bit transparent. That includes all the command-line arguments and all the data and characters interpreted by the commands.

Note:

There are, however, still certain restrictions on the portability of 8-bit data between SINIX and other systems.
- Data interchange between systems over email links may be restricted to 7-bit data on

account of the mail or networking protocols being used.
- 8-bit data and file names may only be portable to systems which comply with the X/Open system interface definition (see "X/Open Guide, System Interface Definitions" [13]).
- To ensure data portability to all POSIX-compliant systems (Portable Operating System Interface for Computer Environments), the characters used in file names should be limited to those in the Portable Filename Character Set, which comprises the following characters (see "X/Open Guide, System Interface Definitions" [13]):

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
```

2.1.2 Multibyte characters

In Asiatic and other languages, the number of symbols known as ideograms that need to be represented is too great to be handled even by 8-bit coding. For example, the full set of Chinese symbols comprises over 65,000 ideograms, far too many to allow unique 8-bit coding. In this case each symbol needs to be represented by multiple bytes, i.e. byte sequences of 16 or even 32 bits. Byte sequences of this type used to represent a single character are known as multibyte characters.
Every ideogram that needs to be represented is coded as a sequence of bytes and is interpreted by the associated operating system, program and terminal as a single ideogram. The length of ideogram coding does not have to be uniform: the multibyte coding for a character set may include characters coded as 8-bit characters (single-byte codes), with single-byte coding being viewed as a special case of multibyte coding. The only condition to which multibyte character coding is subject is that a multibyte character may not contain a null character as one of its bytes.
The current situation with regard to the coding of ideographical scripts is that hardware and software producers have set up a variety of multibyte codes as national standards. Following wide-ranging efforts an initial version of ISO Standard 10646 was agreed in 1992. It defines "Universal Coded Character Sets (UCS)" which also incorporate the UNICODE codesets.

2.1.3 Wide characters

Multibyte character working is made complicated by the fact that characters and ideograms are not uniformly coded with an identical number of bits or bytes. ANSI/ISO C supplies a new internal variable type `wchar_t` which allows multibyte characters with different codings to be treated as data objects of uniform size, known as wide characters. `wchar_t` is a `typedef` that is declared in the `stdlib.h`. header file *(*see "C-DS C V 1.0 Programmer's Reference Manual*"* [6]*)*.
For every wide-character code there is a corresponding multibyte code and vice versa. The wide-character code for a character in the portable character set must have the same value as the character in the character set; and the same applies to the null character *(*see "X/Open Guide, XPG3-XPG4 Base Migration Guide" [11]*)*.
The portable character set comprises the following characters (see "X/Open Guide, System Interface Definitions" [13]):

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

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 ! # % ^ & * ( ) _ + - = { } [ ]
: " ~ ; ' ' < > ? , . | \ / @ $
```

2.1.4 Multibyte capability

Programs with multibyte capability must be 8-bit transparent; in other words, they must not use the 8th bit of a byte for validation purposes or to flag internal states.
In addition, internal processing of program data must be based solely on wide characters. This data type ensures that every possible character from every possible character set can be incorporated in the coding; so no part of the coding will be lost.
To create unrestrictedly portable programs it is subsequently necessary to subject the wide characters to processing by multibyte functions. Multibyte functions provide the same functionality as standard C functions but are also capable of processing wide-character values (see "Multibyte functions").

2.2 Collating sequences

Software programs often need to have information about the order in which characters are to be sorted (collated). This is significant, for example, when listing the files in a directory, when outputting sorted lists or names and numbers from a database, or when generating an index for a book.
The collating sequence defines the value and position of each character relative to the other characters in a character set. Different languages use different collating sequences, and this has to be taken into account in software program design.
In the ASCII code table, for example, the binary coding of the characters follows the alphabetical order of the English language. Thus it is possible to sort words using relatively simple algorithms to compare binary values. However, with other code tables, such as the ISO 8859-1 character set (languages in the Western European sphere), the task is more complicated, as the coding sequence is not the same as the collating sequence and the same characters collate differently in different languages.
Hence sorting information must not be made an integral part of the coding of a program's product core. It should be held in collating sequence tables matched to national requirements and dynamically accessible by appropriately designed algorithms.
For further information on collating sequences refer to the chapter "Developing internationalized software".

2.3 Character classification

Another significant aspect of character string processing is the ability to classify the characters of any character set in groups. This makes it possible to establish whether a given character code belongs, for example, to the class of uppercase letters, lowercase letters, decimal digits (0-9), punctuation characters, printing characters, control characters or space characters.
This information is used by character classification functions, string comparisons and internationalized regular expressions. The functions are also used for character class definition (see chapter "Developing internationalized software") and for character conversion, for example upshifting and downshifting (lowercase to uppercase and vice versa).
Like collating sequences, character classes should also not be made part of the coding of product core of a program which is intended to be portable. Character class information can be defined for every character set in character classification and shift tables dynamically accessible at program runtime.
For further information on character classification and up/downshifting refer to the chapter "Developing internationalized software".

2.4 Local customs

The aspects discussed so far relate primarily to language and thus to the characters and character sets that are used. Another significant aspect of internationalization is the question of conventions for representing cultural data appropriate to a country or a region. This information must likewise be kept out of the product core coding. It needs to be stored in dynamically accessible national databases.

2.4.1 Numeric information

The conventions for representing numeric and currency-related information vary from one country to another. In the United States and Great Britain, for example, the radix character (which separates the integer part of a number from the fractional part) is a dot (.) and the digit grouping symbol (which separates groups of thousands in large numbers) is a comma (,). In some European countries, such as France and Germany, the usage of these symbols is reversed, with the dot acting as digit grouping symbol and the comma as radix character. There is even more variety in the representation of monetary values. As with other numeric values, every country has its own usage rules for the radix character and the digit grouping symbol; and in addition a whole range of different symbols (e.g. $, ¥, £), letters (e.g. DM, FF, kr) and combinations of the two (e.g. £m) are used to identify the type of currency. The positioning of the currency symbol varies as well: it precedes the numeric value in the United States, follows it in Germany and acts as the radix character in Portugal.

2.4.2 Date and time handling

Every language naturally has its own names for days of the week and months of the year, and software products need to be able to represent these names. In addition there are local customs to take into account in representing a specific day of a year. Thus the numeric day and year specifiers may be combined with the name of the month and the day of the week, or abbreviated forms of the names may be used, or the date may be given in numeric form only. The order of the components also varies from one country to another.
Furthermore, date handling depends on the type of calendar in use. If you live in a country which uses the Gregorian calendar, the calendar year has 365 days (366 in leap years) and 12 months. Dating is based on the traditional year of the birth of Christ, and each year begins on January 1. By contrast in Israel the year has 12 months and between 353 and 355 days, while leap years have 13 months. There is a similar situation in many Islamic states, where the year has 12 months and 354 or 355 days.
The conventions for representing the time are as various as those for the date. In the United States, for example, the time is given in 12-hour notation with an am (*ante meridiem*) or pm (*post meridiem*) suffix, while in most other countries the 24-hour clock is used. There are also differences in the separators between the hours, minutes and seconds. Then there are timezones to be taken into account, as well as daylight saving time (summer time) arrangements in some countries.

3 Developing internationalized software

The term *internationalization* ideally refers to the process of designing software without making assumptions about the language, code tables, or national and local conventions of the environment in which the program will be used. On system level that means that internationalized programs link in the country-specific information they need from national databases at runtime using specially designed interfaces.
The program design process for internationalized applications has to allow for a number of fundamental principles. Internationalized programs are typically designed as two separate components. One component is a neutral program core containing the functional code and capable of being used unmodified anywhere in the world. The other consists of national databases containing information specific to the place of use. In order to keep the two components fully separate these databases must be accessible only via neutral function interfaces. For detailed information on the fundamental principles of internationalized program design refer to "IHB - Internationalization Handbook" [8].
On SINIX V5.41 systems the modular integration of country-specific information is handled by the *Native Language System* (NLS), which was designed by the X/Open Group. The NLS supplies C programmers writing for UNIX/SINIX with a programming interface which allows code to be written independently of assumptions about language, code tables and national conventions (refer to "X/Open Guide, Internationalisation Guide" [14] and "X/Open Guide, System Interface Definitions" [13]).
The Native Language System (NLS) provides the following functionality:

- internationalized C library functions which make no assumptions about native language, country or character set and are thus suitable for handling universal character classification, case conversion (up/downshifting), number format conversions and string sorting (collation).
- a set of C library functions which allow a personal language environment to be set, modified and deactivated at application program runtime.
- an announcement mechanism which allows users to specify their own native language, local custom and codeset preferences at application runtime. These settings are made with the aid of environment variables which are associated with specific categories within national databases.
- message catalogs which allow the messages for an application program to be kept separate from the program logic, translated into different languages and bound to the application at runtime.
- 8-bit transparent standard commands capable of processing non-ASCII character sets.

The following sections provide detailed descriptions of the various NLS components needed to develop an internationalized program.

3.1 Country-specific information

Country-specific information in internationalized software is kept separate from the neutral program core in the form of components of national databases known as *locales*. A locale is a working environment, similar to the ordinary shell environment, which users can set in accordance with personal requirements. A locale groups together the cultural conventions which apply to a given country or region, or the personal preferences of a specific user.

However, a locale cannot be equated with a language, as it defines a far broader environment.
Programs require two types of country-specific information:

- information which is specific to one program, such as the message texts output by the program.
  Program-specific information is handled by a messaging system (see section "Message catalogs (LC_MESSAGES category)" and section "Generating message catalogs").
- information which is program-independent, such as the format for date strings in a particular language.
  The program-independent information, often also referred to as a locale, is held in *NLS databases* (see chapter "Localizing internationalized software"). These databases are not databases in the normal sense of the word; they are simply collections of specific items of country-specific data made up of the following types of information:
  - configuration data
  - codeset descriptions and coding information
  - character classification tables
  - collating sequence tables
  - shift tables
  - language-specific and country-specific information

3.2 Locale and categories

The NLS allows both for different native languages and for country-specific features. The elements of a locale as stored in the NLS databases break down into various subgroups known as *categories*. These provide flexible selective access to defined areas of NLS information.
Each category is associated with an identically named environment variable which is evaluated whenever an environment is initialized by the `setlocale()` function.
A locale can comprise the following categories:

- the LC_COLLATE category, which handles collation information. It comprises tables defining the collating sequence for each supported native language. The following capabilities are supported to make it possible to collate characters and character combinations other than those in the ASCII character set:
  - 1-to-1 character mappings
  - 1-to-2 character mappings
    Here certain characters are treated as if they were two characters (such as the German *ß*, which is collated as *ss*).
  - n-to-1 character mappings
    Here certain character sequences are treated as if they were just one character (such as the Spanish consonants *ch* and *ll*, which are collated after *c* and *l* respectively).
  - "don't care" character collation
    For example, if the hyphen '-' were defined as a don't care character, the strings *re-locate* and *relocate* would be viewed as identical.
- the LC_TYPE category, which handles character classification and conversion (shifting).
  The character classification tables assign specific attributes to the individual characters in the character set, such as whether a character is printable, or whether it is a number or a letter (German and Greek classify ß as a letter; many other countries do not).
  The corresponding C macros (see table "Accessing categories from functions ") access this locale-dependent information.
  The character conversion information controls shifting from lowercase to uppercase and vice versa (upshifting and downshifting). For each selectable character set there is an internal table defining the effects of shifting. In Hungarian, for example, the uppercase equivalent of lowercase *é* is *É*, while in French it is *E* (with no accent).
- the LC_MESSAGES category, which covers all textual information such as affirmative and negative answers to yes/no queries.
- the LC_MONETARY category, which covers formatting conventions for numeric values such as currency information.
  In addition to defining the digit grouping symbol and the radix character, this category specifies the nature and positioning of the currency symbol, which has a major impact on accounting programs, among others. Thus the currency symbol for the US dollar precedes the amount (e.g. $1,000.00), while the symbol for the Portuguese escudo acts as the radix character in the amount (e.g. 1.000$00).
- the LC_NUMERIC category, which covers numeric information relating to the form of the radix character, the digit grouping symbol and the exponentiation sign.
  In English-speaking areas the radix character takes the form of a dot (e.g. 2.15), while in German-speaking areas a comma is used instead (e.g. 2,15). In English-speaking areas the digit grouping symbol is a comma (e.g. 1,350,000), while in German-speaking areas it is a dot (e.g. 1.350.000).
- the LC_TIME category, which covers date and time information (including day and month

names). The order in which the day, month and year elements of the date are given and the separators used between them vary from country to country.
Thus October 7, 1992 is represented as 10/7/1992 in the United States, 1992/10/7 in Japan, and 7.10.1992 in Germany. In the United States the time is given in 12-hour notation with an am (*ante meridiem*) or pm (*post meridiem*) suffix, e.g. 07:43 pm, while Germany uses the 24-hour clock, e.g. 19.43.
Names of the months and of the days of the week and the usual abbreviations for them are also covered by the LC_TIME category.

3.3 Single-byte and multibyte functions

Under UNIX/SINIX there are two types of internationalized program that can be created. If you internationalize using single-byte functions, i.e. C functions with 8-bit capability, the programs you create will not be able to process multibyte characters. This ability is not really necessary for internationalized programs intended for America and Europe, as there the ISO Standard 8-bit character sets are sufficient. By contrast, if you create internationalized programs using multibyte functions, your programs will also be capable of processing single-byte characters.
The following sections list all the single-byte and multibyte functions available on SINIX V5.41 systems.

3.3.1 Single-byte functions

The following single-byte functions have been defined or enhanced as part of the NLS in order to support internationalized program development:

atof( )	catclose( )	catgets( )	catopen( )	fprintf( )
fscanf( )	gcvt( )	getdate( )	isalnum( )	isalpha( )
iscntrl ( )	isdigit ( )	isgraph( )	islower( )	isprint( )
ispunct( )	isspace( )	isupper( )	isxdigit ( )	localeconv ( )
nl_langinfo( )	printf( )	scanf( )	setlocale( )	sprintf( )
sscanf( )	strcoll( )	strftime( )	strtod( )	strxfrm( )
tolower( )	toupper( )	vfprintf( )	vprintf( )	vsprintf( )

Table 5: Single-byte functions

The `getdate()` function is available only on SINIX V5.41 systems. In section "Appendix A: Interface classification" you will find details of the origins of the named interfaces. For further information on these functions refer to "C-DS C V1.0 Programmer's Reference Manual" [6].
The following header files are used to supply definitions and declarations for NLS-aware programs:
ctype.h
langinfo.h
limits.h
locale.h
nl_types.h
For further information on these header files refer to "C-DS C V1.0 Programmer's Reference Manual" [6].
The following commands are useful for program internationalization purposes:

dumpmsg	extract	gencat
iecho	iput	

Table 6: Internationalization commands

For further information on these commands refer to the SINIX V5.41 manuals "Commands Volumes 1-3" [1-3].

3.3.2 Multibyte functions

The C functions described above were designed to handle characters requiring no more than 8 bits and are thus of limited use for processing multibyte character sets. The ideal solution would have been to enhance these C functions to handle wide characters; but for reasons of compatibility with earlier systems this is not possible.
Hence the approach adopted was to define a parallel set of C functions known as multibyte functions (Worldwide Portability Interfaces/WPI, see "X/Open Guide, Internationalisation Guide" [14]) with functionality equivalent to that of the standard C functions but also capable of processing wide character values. Thus if you are required to write unrestrictedly portable applications you should use the multibyte functions listed below.
For further information on these functions refer to "C-DS C V1.0 Programmer's Reference Manual" [6].

fgetwc( )	fgetws( )	fputwc( )	fputws( )	getwc( )
)	getws( )	)	iswalpha( )	iswcntrl( )
iswdigit( )	iswgraph( )	iswlower( )	iswprint( )	iswpunct( )
)	iswupper( )	iswxdigit( )	mblen( )	mbstowcs( )
mbtowc( )	putwc( )	putwchar( )	putws( )	towlower( )
)	wcscat( )	wcschr( )	wcscmp( )	wcscoll( )
wcscpy( )	wcscspn( )	wcsftime( )	wcslen( )	wcsncat( )
wcsncmp( )	wcsncpy( )	wcspbrk( )	wcsrchr( )	wcsspn( )
wcstod( )	wcstok( )	wcstol( )	wcstombs( )	wcstoul( )
wcswcs( )	wcswidth( )	wcxfrm( )	wctomb( )	wcwidth( )

Table 7: Multibyte functions

3.4 Accessing categories from functions

The table below indicates which of the above functions you can use to access the various NLS categories.
Note that the functions only operate as described if they have first been initialized with the `setlocale()` function (see section "Initializing an internationalized program").

Category	Function group
LC_CTYPE	`tolower()`, `toupper()`, `isalpha()`, `isupper()`, `islower()`, `isalnum()`, `isspace()`, `ispunct()`, `isprint()`, `isgraph()` and `iscntrl()`
LC_COLLATE	`strcoll()` and `strxfrm()`
LC_MESSAGES	`catopen()`, `catgets()` and `catclose()` Access to yes/no information with the `nl_langinfo()` function
LC_MONETARY	`localeconv()` with appropriate parameters
LC_NUMERIC	`printf()` and `scanf()` family (`fprintf()`, `fscanf()`, `sprintf()`, `sscanf()`, `vfprintf()`, `vprintf()`, `vsprintf()` and `gcvt()`, `strtod()` and `atof()` conversion functions, Access with `nl_langinfo()` plus appropriate parameters
LC_TIME	`getdate()` and `strftime()` Access with `nl_langinfo()` plus appropriate parameters

Table 8: Accessing categories from functions

The `nl_langinfo()` function is a general query function used to look up information about a specific locale (see "C-DS C V1.0 Programmer's Reference Manual" [6]). In the following example the word "ja" is displayed in a German locale, "oui" in a French locale.

Example:

```
yes.c
#include <stdio.h>
#include <locale.h>
#include <langinfo.h>
main ( )
{
(void) setlocale (LC_ALL, "");
printf("%s.\n", nl_langinfo(YESSTR));
exit(0);
}
$cc -o yes yes.c
$LANG=invalid yes
yes.
$LANG=De_DE.88591 yes
ja.
```

```
$LANG=Fr_FR.646 yes
oui.
```

3.4.1 Character classification (LC_CTYPE category)

One major aspect of the way an application works is its approach to classifying characters. The character classes within a character set (see section "Regular and internationalized regular expressions") are:

- uppercase letters
- lowercase letters
- decimal digits (0-9)
- punctuation characters
- control characters
- whitespace characters (blanks and tabs)
- hexadecimal digits

For each character in a defined character set there is an entry in the character classification table for the related language. Each entry in this table contains a series of flags identifying the truth or falsehood of a particular language assertion.
In the past, applications classified characters according to whether their value fell within a given numeric range. If, for example, you were using the ASCII character set, you could use the following test to search for all uppercase letters:

```
if(c>='A'&&c<='Z')
```

This test worked because all the uppercase letters in the ASCII character set have values in the range between 0x41 and 0x5a (A-Z). However, this does not apply to the ISO 8859-1 character sets accepted in internationalized programs, where uppercase letters are assigned to the ranges 0x41 - 0x5a, 0xc0 - 0xc6 and 0xd8 - 0xdf (refer to the "X/Open Guide, Internationalisation Guide" [14]).
So for all character classifications in internationalized applications you should use only the internationalized `ctype` functions (see "Single-byte functions " and "Multibyte functions "). These functions classify values on the basis of the `type` information in the application's locale (LC_CTYPE category, covering all character classification information) and are therefore independent of language and character set.
If you use the `ctype` single-byte functions, the test for uppercase letters is as follows:

```
if(isupper(c))
```

and if you use the multibyte functions:

```
if(iswupper(c))
```

For further `ctype` character classification functions refer to the table "Accessing categories from functions ".

3.4.2 Character conversion (LC_CTYPE category)

Like character classification, character conversion (shifting) depends on the type of character set being used. With ASCII characters, the expression

letter `|=0x20;`

is sufficient to downshift *letter*, i.e. convert it to its lowercase equivalent.
Similarly the expression

letter `&=0xdf;`

will upshift *letter*, i.e. convert it to its uppercase equivalent, as the uppercase letters in the ASCII character set occupy the values between 0x41 and 0x5a (A-Z), while the lowercase

letters are in the range between 0x61 and 0x7a (a-z).
This method of character conversion must not be used in internationalized applications, as it only works for ASCII characters, and the validity of the input is not checked.
Thus in international applications you should use only the internationalized `ctype` functions `tolower()` and `toupper()` to perform downshifting and upshifting. Using the single-byte functions the correct expressions for these operations are

letter `= tolower(`*letter*`);`

for downshifting and

letter `= toupper(`*letter*`);`

for upshifting.
These two functions use information from the application's locale (LC_CTYPE category, covering all character conversion information) and are therefore independent of the character set. In addition they check whether the input is valid, i.e. either an uppercase letter or a lowercase letter, and return the argument unmodified if the input is invalid.
In a multibyte environment the towlower and towupper functions must be used.

Note:

When performing character comparisons and assignments, note that the C variable type `char` has an implicit sign. To avoid unintended side-effects you should always define characters as `unsigned char`.

Processing multibyte characters

Multibyte characters in the C programming language are often stored externally as byte strings of type `char`, and this may cause problems in terms of internal processing. When processing objects of this data type internally it is often useful to first convert them to wide-character codes of type `wchar_t`, then process them internally, and finally convert them back to multibyte sequences for output.
Conversion can be done with the following C functions:

`mblen()`	return length of multibyte character
`mbtowc()`	convert multibyte characters to wide characters
`wctomb()`	convert wide characters to multibyte characters
`mbstowcs( )`	convert multibyte string to wide character string
`wcstombs( )`	convert wide character string to multibyte string

Table 9: Multibyte character conversion functions

Example

The `mbtowc()` function is typically used in the following way:

```
char *mb;
wchar_t wchar;
...
mbtowc(&wchar, mb, strlen(mb));
```

The `mb` argument points to a multibyte character. After execution of `mbtowc()` the `wchar` variable contains the corresponding wide-character code (see "C-DS C V1.0 Programmer's Reference Manual" [6]).

3.4.3 String comparison (LC_COLLATE category)

String comparisons in an internationalized environment are based on the relative positions of the characters, which are governed by the language and character set being used. For a number of languages the UNIX/SINIX operating system must provide *multipass* sorting algorithms, These algorithms are processed in a series of passes on the basis of set priorities such as to

- arrange diacritics (such as accents) within a character class (e.g. a, â, á, à).
- collate *n*-character sequences as single characters (in Spanish, for example, collating ch after the letter c)
- collate single characters as 2-character sequences (in German, for example, collating ß as ss)
- ignore characters (such as the hyphen in dictionary entries) in the collating sequence

Hence in internationalized applications it is no longer possible to make direct use of the `strcmp()`, `strncmp()` and `memcmp()` functions.
Instead you can now either utilize the `strcoll()` function or use the `strxfrm()` function to transform the strings to a type to which the `strcmp()` or `strncmp()` function can be applied.
There are also two functions for wide character string comparison which make use of the collating sequence information of the application's locale (LC_COLLATE category):

- the `wcscoll()` function, which processes wide characters using locale-specific collating sequence information, and
- the `wcsxfrm()` function, which transforms wide character strings on the basis of the locate-specific collating sequence information. The transformed strings can then be compared using the `wcscmp()` function.

3.4.4 Message catalogs (LC_MESSAGES category)

The LC_MESSAGES category covers all text-based information.
When working with internationalized programs you communicate with them in your native language, or rather in the language you have set for your personal locale.
The message text strings of an internationalized program are not in the program code; they are kept apart in separate message catalogs. There is a separate message catalog for each selectable native language. When an internationalized program is executed, the message catalog associated with the native language set in the current locale is bound to the program (see section " Defining the working environment").
A strict distinction is made between two types of message text that programs require:

- program-independent replies to messages expecting a yes/no answer in a particular language. These replies are stored directly in the LC_MESSAGES category in the form of values for the statements YESSTR and NOSTR (or their synonyms YESEXPR and NOEXPR).
  For example, if you enter the `rm` command with its `-i` option, you are expected to give an affirmative reply for each file before the command actually deletes it. Depending on what string is defined for the YESSTR statement in the LC_MESSAGES category, you might answer by entering `yes` or `y` in an English locale or `ja` or `j` in a German one.
- program-specific message texts issued by a specific application.
  These program-specific messages are handled by a messaging system (see section "Generating message catalogs").
  The SINIX V5.41 operating system supports two types of program-specific message

catalog:

- message catalogs complying with the messaging interface defined by X/Open
- message catalogs complying with the messaging interface supplied with AT&T System V Release 4.0

Although both messaging systems are part of the SINIX V5.41 base system, the following description is restricted to the X/Open messaging system, which is recommended because it offers a flexible and portable interface. The X/Open messaging system provides for variable storage of message catalogs and allows access to them to be controlled with the NLSPATH environment variable.

There is a description of the AT&T messaging interface in section "Appendix C: Message catalogs as in AT&T System V, Rel. 4.0".

The LC_MESSAGES category defines the language and hence the message catalog containing the program-specific messages texts. The X/Open messaging function `catopen()` opens the required message catalog. As the names of the message catalogs and the directories in which they are stored are variable, the NLSPATH environment variable has been introduced to control message catalog access.

NLSPATH is evaluated by the `catopen()` function. It identifies the storage location of the message catalogs in the form of a search path. The `catopen()` function also interprets metacharacters (see the table "Metacharacters in the NLSPATH environment variable") which help to specify a particular message catalog. The NLSPATH environment variable is normally set up on a system-wide basis and usually does not then need to be modified by individual users.

When generating message strings for internationalized programs, note that the language of the messages is not the only thing that may vary. Some languages also expect different syntax for message string design. Thus the following statement

```
printf(format, weekday, month, day, hour, min);
```

produces a language-independent date and time format.

In an American locale *format* could be a pointer to the string

```
"%s, %s %d, %d:%.2d\n"
```

producing the following message text:

```
Tuesday, October 20, 10:02
```

To produce the equivalent text in a different language, the format string is used as an index to a message catalog and the appropriate translation is retrieved at program runtime.

The argument list in the above example defines a fixed sequence for the substitution arguments. Translating this message text into German, for example, would be something of a problem, as the arguments need to be printed in a different order in keeping with the different national conventions.

This problem cannot be resolved by the use of message catalogs alone. Therefore the functions of the `printf` and `scanf` family have been enhanced to allow a substitution to be applied to the *n*th argument within an argument list, not just to the next unused argument.

To make use of this facility the `%` variable can be replaced by the character sequence `%`*digit*`$`, where *digit* stands for a decimal value identifying the position of the argument in the argument list.

In a German locale *format* could be a pointer to the string

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the following output:

```
Dienstag, 20. Oktober, 10:02
```

The `%`*digit*`$` specifier allows formatted output to be produced in any language and in keeping with any cultural conventions.

Note in connection with this example that date output here merely serves as an illustration of the way syntax varies from language to language. In practice the `strftime()` function provides a simpler and more flexible international program interface for date and time output.

Note for `yacc` programmers
If you call `yacc` with the -i option, the messages of the program code that `yacc` generates are supplied with function calls which provide access to the internationalized message catalogs. The option allows `yacc` programmers to directly generate internationalized programs.
For further information refer to the description of `yacc` in "C-DS C V1.0 Programmer's Reference Manual" [6].

3.4.5 Currency information (LC_MONETARY category)

All currency information is covered by the LC_MONETARY category. All information in this category is program-independent and is located in the NLS databases.
Using the `localeconv()` function you can retrieve numeric formatting information from the current locale. This function and the `lconv` structure are defined in the `locale.h` header file. The `localeconv()` function returns a pointer to a structure of type `struct lconv` containing formatting information for numeric values (monetary and otherwise). A query might take the following form:

```
struct lconv *buffer
.
.
.
buffer = localeconv();
.
.
.
printf("The international currency symbol is '%s'.\n",
buffer->int_curr_symbol);
```

Thus a currency symbol retrieval operation returns different formats as appropriate to the current locale. The following are examples of how 100 units of various currencies are represented.

Example
in the United States:$ 100.00
in Germany:100,- DM
in France:100,00 FF

3.4.6 Numeric information (LC_NUMERIC category)

The LC_NUMERIC category covers the rules and symbols for formatting all numeric information other than currency data. All the information in this category is program-independent and is located in the NLS databases.
The behavior of commands and functions which process decimal and floating-point numbers or convert decimal and floating-point numbers to and from strings is governed by the LC_NUMERIC category.
The form of the radix character, for example, depends on the current locale and is governed by the value of RADIXCHAR in the LC_NUMERIC category.

Example
Form of the radix character:
in the United States: 3/2 = 1.5
in Germany: 3/2 = 1,5
Form of the digit grouping symbol:
in English-speaking areas: 1,000,000.00
in German-speaking areas: 1.000.000,00

3.4.7 Date and time handling (LC_TIME category)

Using the `nl_langinfo()` function and suitable parameters (see the table "Mandatory date and time parameters ") you can retrieve information on how months, days and the appropriate date and time formats are output in the current locale. The expression
`nl_langinfo(MON_1)`
in a French locale, for example, returns `janvier`.
The `strftime()` function converts the date and time to a string, making allowance for the current locale and the format passed to it in its parameters. It converts the internal form of the date and time (the number of seconds since January 1, 1970) to a format appropriate to the locale. This function should always be used prior to date output operations.
The function call
`strftime(str, strsize, "%A, %d. %B %Y", tmptr)`
issued on Thurday, October 29, 1992 in a German locale, for example, would place the string "Donnerstag, 29. Oktober 1992" in `str`.
The `getdate()` function (available on SINIX V5.41 systems only) derives a `tm` structure from a formatted date string. The `tm` structure declaration is in the `time.h` header file (see SINIX V5.41 "C-DS C V1.0 Programmer's Reference Manual" [6]).
With the DATEMSK environment variable set to the name of a file containing the lines `%A, %d, %B, %Y`, the function call
`getdate("Thursday, October 29, 1992")`
returns a pointer to a `tm` structure representing this date.

3.5 Initializing an internationalized program

At runtime, an internationalized program looks up its environment in the NLS database and retrieves the information it requires about its locale. The first step in this process is to provide access to the NLS database. This is controlled by the `setlocale()` function. Except for the `catopen()`, `catgets()` and `catclose()` functions, which form the basis of the X/Open messaging system, all the other C functions which play a part in internationalization do not have their NLS functionality activated until `setlocale()` has been invoked.
The `setlocale()` function can perform three tasks:
- query information about the current locale.
- set the program to use the values of the environment variables.
  There is a list of these variables in section " Defining the working environment"
- set a fixe locale.

The `setlocale()` function is defined in the <*locale.h*> header file. Its syntax is as follows:
`char *setlocale(int` *`category`*`, const char *`*`locale`*`);`
The *category* constant defines whether the entire program locale is to be initialized/queried or only parts of it (specific categories). *category* can have any of the following values:

Constant	**affects:**
LC_ALL	entire locale
LC_COLLATE	behavior of collation functions
LC_CTYPE	behavior of character-handling functions
LC_MESSAGES	behavior of messaging system functions
LC_MONETARY	access to currency information using `localeconv()`
LC_NUMERIC	radix character for I/O and conversion functions
LC_TIME	behavior of date and time functions

Table 10: Areas affected by categories

The *locale* argument is a pointer to a string comprising the settings required by *category* in the form of the name of a defined locale. The values accepted for *locale* depend on the current application. The values described below are valid for all SINIX V5.41 systems.

Querying information about the current locale

`(char *) NULL`
This setting tells `setlocale()` to query and return information about the current locale.

Setting the program to use the values of the environment variables

""
This setting tells *category* to take the value of the appropriate environment variables. This special version of `setlocale()` can be used to create internationalized programs.
In this case `setlocale()` takes the name of the new locale for the specified category from one of the following environment variables:
- $LC_ALL if set, otherwise
- the environment variable of the appropriate category (e.g. $LC_COLLATE) if set, otherwise

- $LANG

Setting a fixed locale

`"C"`
This is the minimum locale for C compilation. Regardless of whether or not there is an NLS database named "C", the locale defaults to the "conventional" UNIX environment of American English and the ASCII character set. This is the default setting for the SINIX operating system and acts as a fallback for internationalization functionality.
`locale-name`
Name of a directory under `/usr/lib/locale`. A fixed value for *locale-name* can be used to create a locale-specific program, i.e. a national variant. To find out which locales are available on the SINIX V5.41 operating system turn to section "Mandatory strings by category ").

Example

There are basically three ways of setting a program locale with the `setlocale()` function:

1. Setting all categories
   To set all categories to reflect a German locale, for example, you use the statement:
   ```
   setlocale(LC_ALL, "De_DE.88591");
   ```
   In this example the program is assigned the locale "De_DE.88591" and all the categories it contains. If *string* is not a valid locale name, `setlocale()` returns a null pointer and leaves the locale unchanged.
2. Setting one category
   In addition to setting all categories it is possible to select just one of the categories LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC or LC_TIME.
   ```
   setlocale(LC_COLLATE, "En_GB.88591");
   ```
   In this example `setlocale()` selects the LC_COLLATE category (containing collating sequence information) from the locale "En_GB.88591", thus telling the program to use a British collating sequence.
3. Setting a modifier
   If you need to define a category even more precisely, you have the option of adding a modifier to the locale specification.
   ```
   setlocale(LC_COLLATE, "De_DE.88591@TE");
   ```
   In this example `setlocale()` uses the "@TE" modifier to select a collating sequence which causes a program to sort on the basis of the collating sequence used in the German telephone directory, even if there is a different collating sequence defined in the environment variables.

3.6 Internationalizing an existing program

To internationalize an existing program you need to apply all the techniques discussed in the preceding sections. The only difference is that you do not have to write all the code from scratch but can simply modify the existing source. The following list once more summarizes the principal programming techniques that you have to bear in mind when developing internationalized programs and internationalizing an existing program.

- **Logical operations, masks, comparisons, assignments**
  The most significant bit of a byte is meaningful in an 8-bit environment and must not be used to flag internal states of an application (see section "Character sets").
  It is also essential to check whether there are any logical bit operations which modify data bytes.
  When performing character comparisons and assignments, note that the C variable type `char` has an implicit sign. To avoid unintended side-effects you should always define characters as `unsigned char` (see section "Character conversion (LC_CTYPE category)").
- **`ctype` macros**
  When classifying characters you should only use the `ctype` functions listed below (see section "Character classification (LC_CTYPE category)").
  Single-byte functions:
  `isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit()`
  Multibyte functions:
  `iswalnum(), iswalpha(), iswcntrl(), iswdigit(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit()`
  When shifting characters you should only use the following `ctype` functions (see section "Character conversion (LC_CTYPE category)"):
  Single-byte functions:
  `tolower(), toupper()`
  Multibyte functions:
  `towlower(), towupper()`
- **String comparison**
  As the relative order of characters within a character set is governed by the locale, it is no longer possible to make direct use of the `strcmp()`, `strncmp()` and `memcmp()` functions. Instead you can now either utilize the single-byte function `strcoll()` or the multibyte function `wcscoll()`, or use the single-byte function `strxfrm()` or the multibyte function `wcsxfrm()` to transform the strings to a type to which the single-byte functions `strcmp()` and `strncmp()` or the multibyte functions `wcscmp()` and `wcsncmp()` can be applied (see section "String comparison (LC_COLLATE category)").
- **Date and time handling**
  The `strftime()` function converts the date and time to a string, making allowance for the current locale and the format passed to it in its parameters. It should always be used prior to date output operations.
  The `getdate()` function derives a `tm` structure from output formatted with the `strftime()` function (see section " Date and time handling (LC_TIME category)").
- **Numeric information**
  Using the `localeconv()` function you can retrieve and convert formatting information

such as number formats and currency information (see section " Numeric information (LC_NUMERIC category)").

- **`nl_langinfo()` function**
  If you use the `nl_langinfo()` function with valid parameters, you can retrieve any information you need from the NLS database. The expression
  `nl_langinfo(YESSTR)`
  (or YESEXPR), for example, returns the currently applicable affirmative response. There is a list of the valid parameters for this function in the table "Mandatory strings by category " and on the system in the `<langinfo.h>` header file.

When internationalizing an existing program using existing sources it is essential to automate the internationalization process as much as possible. This is especially applicable to message catalogs, as all the messages in the source text have to be replaced with function calls.
The `extract` command which is part of the X/Open messaging system is a parametrizable tool which lets you interactively replace messages from a program's C source file with `catgets()` calls and include the necessary declarations. In the process it automatically creates a suitable message source file. A help file and a pattern file for the `extract` command are stored in the `/usr/lib/nls/extract/` directory.

Example

```
=================
ORIGINAL PROGRAM: tst.c
=================
main()
{
printf("hello world\n");
}
======================
EXTRACT'S OUTPUT FILE: nl_tst.c
======================
#ifdef INTLM     /*** X/Open message handling ***/
#   include <nl_types.h>
nl_catd _m_catd;
nl_catd catopen();
char *catgets();
#else
#   define catgets(a,b,c,d) d
#endif           /*** X/Open message handling ***/
main()
{
printf(catgets(_m_catd, 1, 1, "hello world\n"));
}
============================
RESULT AFTER POST-PROCESSING: nl_tst.c
============================
#   include <nl_types.h>
#   include <locale.h>
nl_catd _m_catd;
nl_catd catopen();
char *catgets();
main()
{
```

```
setlocale (LC_ALL, "");
_m_catd = catopen("tst.cat", 0);
printf(catgets(_m_catd, 1, 1, "hello world\n"));
catclose(_m_catd);
}
```

The above segment of a C program

```
printf("hello world\n");
```

is removed by the `extract` command and replaced with a call to a routine accessing a message catalog in the following format

```
printf(catgets(_m_catd, 1, 1, "hello world\n"));
```

The corresponding message source file is generated automatically:

```
$set 1
$quote "
1 "hello world\n"
```

The message source file now contains the text strings extracted from the C program source code file. After translation these strings act as input for the `gencat()` command.

Example

The example below illustrates internationalization of an existing program. The program generates a welcome screen. It is first shown in a non-internationalized form. Then comes the internationalized source with the English and German message catalogs.
The functions, commands and header files related to internationalization are highlighted in **bold fixed-width font**.
The single-byte functions used in the internationalized source access information from the LC_MESSAGES, LC_TIME and LC_CTYPE categories.

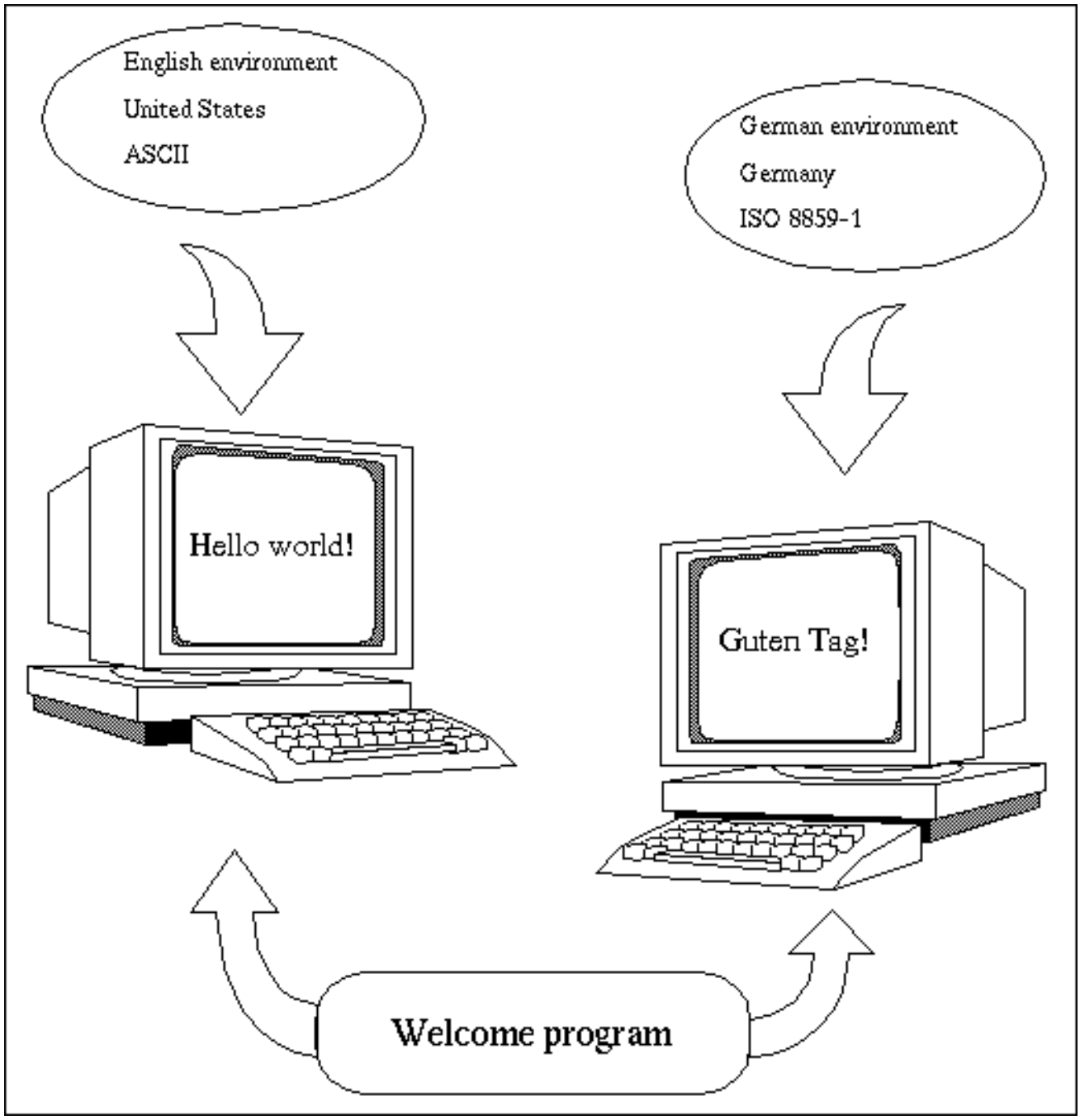

Figure 2: Example for a welcome screen

```
==============================
NON-INTERNATIONALIZED PROGRAM: greeting.orig.c
==============================
#include <stdio.h>
#include <time.h>
int status;
main()
{
time_t clock;
register c, i;
```

```
char ch_name[BUFSIZ], la_name[BUFSIZ];
fprintf(stderr, "Welcome to our automatic greeting program\n");
(void) time(&clock);
fprintf(stderr, "The current time is %s.\n\n", ctime(&clock));
fprintf(stderr, "Please enter your first name : ");
for (i = 0; (c = getc(stdin)) != '\n' && c != EOF; i++) {
if ( (c >= 'a' && c <= 'z') || c == '-')/* Convert the */
ch_name[i] = c;/* first name */
else if (c >= 'A' && c <= 'Z')/* to lowercase */
ch_name[i] = c - 'A' + 'a';
else {
ch_name[i] = '\0';
fprintf(stderr, "Sorry, %s??? ... No name may contain a '%c' !\n",
ch_name, c);
status = 1;
quit();
}
}
ch_name[i] = '\0';
if (ch_name[0] != '-')/* First letter to uppercase */
ch_name[0] = ch_name[0] + 'A' - 'a';
else {
fprintf(stderr, "Sorry, ?%s ... No name may start with a '-' !\n",
ch_name+1);
status = 1;
quit();
}
fprintf(stderr, "Thanks %s, now enter your last name : ", ch_name);
for (i = 0; (c = getc(stdin)) != '\n' && c != EOF; i++) {
if ( (c >= 'A' && c <= 'Z') || c == '-')/* Convert the */
la_name[i] = c;/* last name */
else if (c >= 'a' && c <= 'z')/* to uppercase */
la_name[i] = c - 'a' + 'A';
else {
la_name[i] = '\0';
fprintf(stderr, "Sorry, %s??? ... No name may contain a '%c' !\n",
la_name, c);
status = 1;
quit();
}
}
la_name[i] = '\0';
if (la_name[0] == '-') {
fprintf(stderr, "Sorry, ?%s ... No name may start with a '-' !\n",
la_name+1);
status = 1;
quit();
}
fprintf(stderr, "%s %s, I'm happy to have met you !\n", ch_name,
la_name);
/* There may be some further processing here */
status = 0;
quit();
}
```

```
quit()
{
fprintf(stderr, "Bye, bye !\n");
exit(status);
}
=========================
INTERNATIONALIZED PROGRAM: greeting.c
=========================
#include <stdio.h>
#include <time.h>
#include <ctype.h>
#include <locale.h>
#include <nl_types.h>
nl_catd _m_catd;
nl_catd catopen();
char *catgets();
int status;
main()
{
time_t clock;
register c, i;
unsigned char ch_name[BUFSIZ], la_name[BUFSIZ], time_buffer[BUFSIZ];
(void) setlocale(LC_ALL, "");
_m_catd = catopen("greeting", 0);
fprintf(stderr, catgets(_m_catd, 1, 1, "Welcome to our automatic \
greeting program\n"));
(void) time(&clock);
(void) strftime(time_buffer, BUFSIZ, "%c", localtime(&clock));
fprintf(stderr, catgets(_m_catd, 1, 2, "The current time is %s.\n\n"),
time_buffer);
fprintf(stderr, catgets(_m_catd, 1, 3, "Please enter your first \
name : "));
for (i = 0; (c = getc(stdin)) != '\n' && c != EOF; i++) {
if ( islower(c) || c == '-')/* Convert the */
ch_name[i] = c;/* first name */
else if (isupper(c))/* to lowercase */
ch_name[i] = tolower(c);
else {
ch_name[i] = '\0';
fprintf(stderr, catgets(_m_catd, 1, 4, "Sorry, %s??? ... No name \
may contain a '%c' !\n"), ch_name, c);
status = 1;
quit();
}
}
ch_name[i] = '\0';
if (ch_name[0] != '-')            /* First letter to uppercase */
ch_name[0] = toupper(ch_name[0]);
else {
fprintf(stderr, catgets(_m_catd, 1, 5, "Sorry, ?%s ... No name may \
start with a '-' !\n"), ch_name+1);
status = 1;
quit();
```

```
}
fprintf(stderr, catgets(_m_catd, 1, 6, "Thanks %s, now enter your last \
name : "), ch_name);
for (i = 0; (c = getc(stdin)) != '\n' && c != EOF; i++) {
if ( isupper(c) || c == '-')/* Convert the */
la_name[i] = c;/* last name */
else if (islower(c))/* to uppercase */
la_name[i] = toupper(c);
else {
la_name[i] = '\0';
fprintf(stderr, catgets(_m_catd, 1, 4, "Sorry, %s??? ... No name \
may contain a '%c' !\n"), la_name, c);
status = 1;
quit();
}
}
la_name[i] = '\0';
if (la_name[0] == '-') {
fprintf(stderr, catgets(_m_catd, 1, 5, "Sorry, ?%s ... No name may \
start with a '-' !\n"), la_name+1);
status = 1;
quit();
}
fprintf(stderr, catgets(_m_catd, 1, 7, "%s %s, I'm happy to have met \
you !\n"), ch_name, la_name);
/* There may be some further processing here */
status = 0;
quit();
}
quit()
{
fprintf(stderr, catgets(_m_catd, 1, 8, "Bye, bye !\n"));
catclose(_m_catd);
exit(status);
}
========================
ENGLISH MESSAGE CATALOG: greeting.msg
========================
$quote "
$set 1
1"Welcome to our automatic greeting program\n"
2"The current time is %s.\n\n"
3"Please enter your christian name : "
4"Sorry, %s??? ... No name may contain a '%c' !\n"
5"Sorry, ?%s ... No name may start with a '-' !\n"
6"Thanks %s, now enter your last name : "
7"%s %s, I'm happy to have met you !\n"
8"Bye, bye !\n"
========================
GERMAN MESSAGE CATALOG: greeting.de.msg
========================
$quote "
$set 1
```

```
1"Willkommen in unserem automatischen Begruessungsprogramm\n"
2"Das aktuelle Datum ist %s.\n\n"
3"Geben Sie bitte Ihren Vornamen an : "
4"Es tut mir leid, %s??? ... Kein Name darf ein '%c' enthalten !\n"
5"Es tut mir leid, ?%s ... Kein Name darf mit einem '-' anfangen !\n"
6"Danke %s, geben Sie dann Ihren Nachnamen an : "
7"%s %s, es hat mich gefreut Sie kennenzulernen !\n"
8"Auf Wiedersehen !\n"
```

3.7 Regular and internationalized regular expressions

Regular expressions are a tool for scanning a text for strings which match a defined pattern. Among other things you can check whether a string or a passage of text includes characters from a defined character class.
A regular expression stands for a set of strings. Each member of this set of strings is said to be matched by the regular expression. A pattern is constructed from one or more regular expressions.
A regular expression comprises a string of characters, which can be further classified into:
- ordinary characters, and
- metacharacters.

Ordinary characters consist of all the characters in the character set other than the newline character and the metacharacters. Within a pattern, ordinary character match themselves; i.e. the pattern *abc* will match only those strings which include the character sequence *abc* anywhere in them.
Metacharacters have special meanings and are not interpreted literally.
There are two forms of regular expression:
- simple regular expressions
- extended regular expressions

The syntax of the various forms of regular expression is described in the SINIX V5.41 manuals "Commands, Volumes 1-3" [1-3].
The following table summarizes the commands which support regular expression syntax:

Command	**Regular expression form**
awk	extended
bfs	simple
csplit	simple
ed	simple
egrep	extended
ex	*)
expr	simple
extract	simple
grep	simple
lex	extended
nl	simple
pg	simple
sed	simple
vi	*)

Table 11: Commands

*) The *ex* and *vi* editors support regular expressions which differ slightly from the simple regular expressions described here. These differences are described in the SINIX V5.41 manuals "Commands, Volumes 1-3" [1-3].

Internationalized regular expressions

Simple internationalized and extended internationalized regular expressions use the same elements as their non-internationalized equivalents, but in addition they provide the syntax options in the table below for the construction of expressions within square brackets [...]. To ensure 100% application portability you should not use ranges of the form `[a-z]` in regular expressions. Existing applications may make use of this type of range for collation within a locale, but they are of restricted usefulness in internationalized applications. As you can replace most ranges by specifying the characters themselves or using character classes, you should modify range definitions as illustrated in the following table:

Expression	**Meaning**
[:class:]	(character class expression) Any character of type *class*. The character classes in an internationalized environment are defined by the environment variables LC_CTYPE and LANG.*class* is one of the following *:alpha* any letter *upper* any uppercase letter *lower* any lowercase letter *digit* any decimal digit (0 to 9) *xdigit* any hexadecimal digit (0 to 9, a to f and A to F) *alnum* any alphanumeric character (letter or digit) *space* any character producing white space in displayed text (whitespace characters = blank and tabs) *punct* any punctuation character *print* any printable character (including the characters in *space*) *graph* any character with a visible representation (excluding the characters in *space*) *cntrl* any control character Examples: In the German locale LANG=LC_CTYPE=De_DE.88591 the characters *ä, ö, ü, ß, a, ..., z* match the regular expression `[[:lower:]]`. The character *ê* (e circumflex) does not belong to *lower* or *alpha*, but it is a printable character. In the French locale LANG=LC_CTYPE=Fr_FR.88591 *ê* belongs to *lower* and *alpha*, *ä* does not.
[=c=]	(Equivalence class expression) Any character or collating element defined as having the same relative order in the current collating sequence as *c*. *c* must not be an equal sign = or a right square bracket]. The collating sequence in an internationalized environment is defined by the environment variables LC_COLLATE and LANG.

	Examples: In the German locale LANG=LC_COLLATE=De_DE.88591 the characters *u* and *ü* form an equivalence class. Consequently the characters *u* and *ü* match the regular expression [[=u=]]. In this locale the regular expressions [[=u=]v], [[=ü=]v] and [uüv] are synonyms. In the French locale LANG=LC_COLLATE=Fr_FR.88591 the character e and all its accented forms (ê, ë, è, é) match the regular expression [[=e=]].
[.cc.]	(Collating symbol) Multicharacter collating elements must be represented in this form to distinguish them from ordinary characters. An expression of this type is collated as a single character. *cc* has to be defined as a valid collating element in the current collating sequence. If *cc* is not a valid collating element, [.cc.] is an invalid expression. The collating sequence in an internationalized environment is defined by the environment variables LC_COLLATE and LANG. Example: In the Spanish locale LANG=LC_COLLATE=Es_SP.88591 *ch* is a valid collating element: in Spanish, *ch* is treated as a single letter collating between *c* and *d*. This letter must be represented in the form [.ch.] to distinguish it from the two-letter string *ch* (see also examples for c1-c2.)
c1-c2	Any character from the range between *c1* and *c2* inclusive in the current collating sequence. Unlike in non-internationalized regular expressions, *c1* and *c2* can also be equivalence class expressions [=c=] or collating symbols [.cc.]. Examples: In the German locale LANG=LC_COLLATE=De_DE.88591 the umlauts are ordered as follows in the collating sequence: a, ä, b, c, ..., n, o, ö, p, ..., t, u, ü, v, ... An umlaut form and the associated non-mutated vowel together form an equivalence class (see examples for [=c=]). Character ranges are therefore evaluated as follows: regular expression matching characters [t-v]t, u, ü, v [t-u]t, u [t-ü]t, u, ü [t-[=u=]] t, u, ü [^t-[=u=]]any character other than t, u, ü

	In the French locale LANG=LC_COLLATE=Fr_FR.88591 the regular expression [a-f] is matched by the characters a, b, c, ç, d, e and all accented forms of e, and f. In the Spanish locale LANG=LC_COLLATE=Es_SP.88591 *ch* is a valid collating element: in Spanish, *ch* is treated as a single letter collating between *c* and *d* (see [.cc.]). Ranges are therefore evaluated as follows: regular expressionmatching characters [a-f] a, b, c, ch, d, e, f [b[.ch.]-eg] b, ch, d, e, g

Table 12: Internationalized regular expressions

4 Localizing internationalized software

The functions discussed in chapter 3 primarily relate to developing internationalized software and internationalizing existing software. The other main branch of internationalization is the task of matching software to the specific national, cultural and language-related requirements of its users. This process is known as *localization*.
Localization here denotes the process of establishing information within a computer system specific to the needs of a particular user or user group. The combination of all this information is known as a *locale*.
The term 'locale' denotes a working environment similar to the ordinary shell environment which users can set in accordance with personal requirements. A locale cannot be equated with a language, as it defines a far broader environment (language, character set, cultural data, etc.).

4.1 Creating NLS databases

Different languages and countries have different expectations of the way in which data is processed and represented by the system. The NLS interface allows users to set up personal nationalized environments appropriate to their own requirements. This entails grouping the data specific to a language and a country or region in categories (databases) within a locale named to reflect the defined environment.
Setting environment variables corresponding to the categories activates access to the locale-specific databases and initializes the nationalized or localized working environment.
To generate NLS databases you need to define the data for the individual categories in a set format and announce it (make it known) to the system.
On a SINIX V5.41 system there are two ways of doing this:

1. Directly defining the various categories using the `chrtbl`, `colltbl`, `montbl` and `mkmsgs` utilities to put the categories in the correct format and in their proper place on the system.
2. Using the `ic` compiler available on older SINIX V5.2x systems to generate source definitions for the above commands and an installation script for an entire locale. The `ic` compiler was introduced to enable existing NLS databases from SINIX V5.2x systems to be reused with only minor modifications.

The next section describes direct definition of individual categories. Then section "IC - Compiler for international databases" gives details of how to use the `ic` compiler and how it functions and describes the syntax of the input files it requires.

4.2 Direct generation of NLS databases

- Defining information for the LC_COLLATE category
  The `colltbl` command creates a file named LC_COLLATE which describes the collating sequence for the corresponding database and can be read by the C functions `strxfrm()` and `strcoll()`. For a full description of the specification file required by this command refer to the description of `colltbl` in "SINIX V5.41 System Administrator's Reference Manual" [4].
- Defining information for the LC_CTYPE and LC_NUMERIC categories
  The `chrtbl` command creates two files, one named LC_CTYPE containing all the character classification, shifting and character width information, the other named LC_NUMERIC containing numeric formatting information. For a full description of the specification files required by this command refer to the description of `chrtbl` in "SINIX V5.41 System Administrator's Reference Manual" [4].
- Defining information for the LC_MONETARY category
  The `montbl` command creates a database named LC_MONETARY containing the formatting conventions for monetary quantities for a specific country. For a full description of the specification file required by this command refer to the description of `montbl` in "SINIX V5.41 System Administrator's Reference Manual" [4].
- Defining information for the LC_MESSAGES category
  The way to generate program-dependent message catalogs for dynamic accessing via the LC_MESSAGES category using the `catopen()` function and the NLSPATH environment variable is described in section "Generating message catalogs" (see also section "Message catalogs (LC_MESSAGES category)").
  The only program-independent information in the LC_MESSAGES category comprises the affirmative and negative answers to yes/no questions and the quit string.
  The *Xopen_info* file defines the above responses plus a number of other parameters. The order in which these parameters are stored in *Xopen_info* is shown in the following table:

Parameter	**Default**	**Function**
TIME_FMT	"%H:%M"	Default time format without seconds
DAY_FMT	"%m/%d"	Default short date format without year
FULL_DAY	"%a %b %e"	Default date format without year
YESSTR (YESEXPR)	"yes"	Affirmative reply
NOSTR (NOEXPR)	"no"	Negative reply
QUITSTR	"quit"	Quit string
AR_DA	"%b %d	Default date format for `ar(1)`

TE	%H:%M %Y"	
LAST_ DATE	"%a %b %e %H:%M"	Default date format for `last(1)`
LS_DA TE	"%h %d %H:%M"	Default date format for `ls(1)`, `who(1)`
LS_DA TE2	"%h %d 19%y"	Default date format for `ls(1)` > 6 months
PS_DA TE	"%b %d"	Default date format for `ps(1)`
SU_DA TE	"%m %d %H:%M"	Default date format for `su(1)`
TAR_D ATE	"%b %e %H:%M %Y"	Default date format for `tar(1)`, `cpio(1)`, `pr(1)`

Table 13: Contents of Xopen_info

The `mkmsgs` command compiles an input file containing this information and installs it on the system. For a full description of the syntax refer to the description of `mkmsgs` in the manual "SINIX V5.41 Commands, Volume 2" [2].

Example

German replies to yes/no and quit queries (YESSTR, NOSTR, QUITSTR) plus a number of other parameters are stored in the ASCII file *Xopen_info* in the following format:

```
input file:
%H:%M
%d.%m
%a %e.%b
ja
nein
abbrechen
%b %d %H:%M %Y
%a %e.%b %H:%M
%h %d %H:%M
%h %d  19%y
%d.%b
%d.%m %H:%M
%e.%b %H:%M %Y
```

Then the command

`mkmsgs -o` *input_file* `Xopen_info`

is used to generate the German *Xopen_info* file.

- Defining information of the LC_TIME category

To add information of the LC_TIME category you need to create an ASCII file named LC_TIME containing the values for the various date and time parameters and formats in the required locale. The values must each start on a separate line and they must be in the order shown below.

Parameter	**Function**
ABMON_x	Abbreviated name of *x*th month (*x* varies between 1 and 12)

MON_x	Name of *x*th month (*x* varies between 1 and 12)
ABDAY_x	Abbreviated name of *x*th day of week (*x* varies between 1 and 7)
DAY_x	Name of *x*th day of week (x varies between 1 and 7)
T_FMT	Default time format
D_FMT	Default date format
D_T_FMT	Default date and time format
AM_STR	*ante meridiem* string
PM_STR	*post meridiem* string
DFLT_DATE	Default date format for `date` command

Table 14: Mandatory date and time parameters

The following date and time parameters for the LC_TIME category are not mandatory. If specified, they must each start on a new line, come at the end of the LC_TIME file and appear in the order shown below. Any parameter left undefined must be indicated by the presence of a blank line.

Parameter	**Function**
T_FMT_AMPM	Time format on 12-hour clock with AM_PM
ALT_DIGITS	Alternate digit symbols
ERA	Era definition
ERA_D_FMT	Default era date format
ERA_T_FMT	Default era time format
ERA_D_T_FMT	Default era date and time format

Table 15: Optional date and time parameters

The example on the next page illustrates a German LC_TIME file in which the ALT_DIGITS and ERA parameters are left undefined.

Example

```
Jan
Feb
Mar
Apr
Mai
Jun
Jul
Aug
Sep
Okt
Nov
Dez
Januar
Februar
Maerz
```

```
April
Mai
Juni
Juli
August
September
Oktober
November
Dezember
So
Mo
Di
Mi
Do
Fr
Sa
Sonntag
Montag
Dienstag
Mittwoch
Donnerstag
Freitag
Samstag
%T %Z
%d.%m.%y
%a %d.%h.19%y, %T %Z
AM
PM
%a %e.%b.%Y, %T %Z
%I:%M:%S %p
%EY%mgatsu%dnichi (%a)
%H:%M:%S
%Ex %EX
```

4.3 IC - Compiler for international databases

This section deals in more detail with the syntax of the language that the `ic` compiler (internationalization compiler) on SINIX V5.41 systems understands. The compiler is essentially compatible with the `ic` on SINIX V5.2x systems, which means that database sources from V5.2x versions can be used on V5.41 systems with only minor changes being necessary. `ic` is fully backward compatible, which means that a database developed for a SINIX V5.41 system can be compiled on a V5.2x system and the `ic` available there without the need for any changes. An additional benefit of generating a database with the `ic` compiler is that the syntax of all input is fully validated.
The two versions differ mainly in the procedure for using `ic`. On a SINIX V5.2x system an international database is created from a database source. On a SINIX V5.41 system the internationalization compiler `ic` converts a database source *database.in* into a directory *database* containing the following files:

Xopen_info
A file in the AT&T message text source file format. This file contains information in the LC_MESSAGES category, such as answers to yes/no questions, and information in the LC_TIME category, such as the date format used by the `ls` command (see the table "Contents of Xopen_info").

LC_TIME
A file containing a table of date and time information.

chrtbl
A specification file for the `chrtbl` command (refer to "SINIX V5.41 System Administrator's Reference Manual" [4]) containing information in the LC_CTYPE category relating to character classification and shifting and in the LC_NUMERIC category relating to the form of the radix character, the exponentiation sign and the digit grouping symbol.

colltbl
A specification file for the `colltbl` command (refer to "SINIX V5.41 System Administrator's Reference Manual" [4]) containing information in the LC_COLLATE category relating to the collating sequence.

montbl
A specification file for the `montbl` command (refer to "SINIX V5.41 System Administrator's Reference Manual" [4]) containing information in the LC_MONETARY category relating to currency symbols and monetary formatting conventions.

install_locale
A shell script which performs the following functions:
1. It defines the name of the database (*locale-name*).
2. It transforms the `Xopen_info` file using the `mkmsgs` command (see section "Appendix C: Message catalogs as in AT&T System V, Rel. 4.0") and stores it as a message catalog under `/usr/lib/locale/`*locale-name*`/LC_MESSAGES`.
3. It creates the database files LC_CTYPE, LC_NUMERIC, LC_COLLATE and LC_MONETARY using the `chrtbl`, `colltbl` and `montbl` commands.
4. It stores the database files LC_CTYPE, LC_COLLATE, LC_MONETARY, LC_NUMERIC and LC_TIME in the directory `/usr/lib/locale/`*locale-name*.

The `install_locale` shell script installs a locale named *locale-name*. Only the system administrator is allowed to run this shell script. See chapter "Using internationalized software" for a description of how you can work in a personal environment using this locale.

4.3.1 Introduction

If you are planning to localize an internationalized program for a new market for which the country-specific categories are as yet undefined, you will first need to generate this information in the form of a database for the chosen locale.
To serve its purpose the international SINIX database should contain the following information:

1. Information on the codeset used.
2. Information on the properties of each individual character in the codeset (character classification).
3. Information allowing a program to collate user data correctly.
4. Information on how to perform conversions such as the standard conversions `toupper` and `tolower`.
5. Information which can be accessed by some library functions in order to interpret elements such as date, formats and number representation correctly.

To support database generation, a syntax allowing this information to be entered in a simple manner has been developed, and a compiler has been implemented to interpret it.
The following section describes this syntax and the `ic` compiler.

4.3.2 The database source

The following general considerations were followed in the design of the database syntax:

1. The database source should initially only contain ASCII characters because currently UNIX and hence SINIX systems supporting any codeset other than ASCII are very hard to find. At a later stage, when tools like editors to edit non-ASCII source files are commonly available, this restriction may be dropped, allowing for a more comfortable entry of data into the database.
2. The database source should be free format. This especially means that "whitespace" should have no significance other than as separator for tokens in the input language.
3. Comments should be allowed wherever possible. This is one of the reasons why the C language preprocessor is used in the current implementation: it gives you the freedom to intersperse your source with C-style comments.
4. The source should be as self-documenting as possible.

The source for an international SINIX database consists of two major parts:

- the definition of the codeset used (CODESET section).
- the definition of property tables, collation tables, string tables and conversion tables (DATA TABLE section).

4.3.3 The CODESET section

The definition of the codeset used is introduced by the keyword CODESET or EXTENDED CODESET followed by a name for the codeset.
CODESET implies an eight-bit codeset with the possible extension by some special, noncontiguous sixteen-bit codes named multiletters.
EXTENDED CODESET implies a sixteen-bit codeset with no multiletters.

The name which follows CODESET will become the name of the locale once compilation has succeeded. This name may optionally be given on the command line using the -o option. The command-line name will override the name in the database source.
Following this introduction each code is defined by assigning the code's value to an identifier, which may be used to reference the code from then on. This assignment has the form:
Identifier'='*value_list*[':'*property_list*]';'
The *value_list* is a comma-separated list of values; a value may be given as a C-style character constant, in octal, hexadecimal, decimal or ISO notation, or by giving the name of a previously defined code.
Character codes may be subdivided in two classes: simple and combined codes. There are several restrictions that have to be observed when defining codes in the CODESET section:

1. The list of simple codes must contain all codes from code value 0x0 up to and including the code with the highest value defined. This is necessary because the simple codes are not stored in the international database and the runtime access routines assume the existence of all simple codes for speed reasons. The order of definition is free because all values are sorted into ascending machine collation order when the whole codeset definition is read.
2. The list of simple codes may not contain codes with duplicate code values. This is necessary for the same reason as above.
3. If the codeset is not an EXTENDED CODESET, there may be an arbitrary number of definitions for multiletters. Multiletters need not have contiguous code values and will be sorted in ascending machine collation order.
4. If the codeset is an EXTENDED CODESET, there may be no multiletters and all codes defined are simple codes. (This especially means that rules 1 and 2 apply!)

As the generation of a valid database without the above-mentioned restrictions is not possible, all these conditions are thoroughly checked by the compiler and abort compilation after the CODESET section has been parsed.
The optional properties part in the above code definition serves to assign default properties to a code. If it is not given, the code is assumed to be defined but illegal. (This is useful for languages that do not know about some letters defined in a standard codeset.) Properties are given the form of a comma-separated list of keywords.
There is a third kind of statement allowed in this section: the (re-)assignment of default properties to an already defined code. This statement takes the form:
Identifier':'*property_list*';'
The CODESET section is terminated by the two terminal symbols END '.'.
The use of the '#include' facility provided in the language because of the use of the C preprocessor is strongly recommended as most of the codes considered contain common code (for example ASCII or ISO 646 or ISO 8859) in their lower half and therefore using a common header file will reduce the risk of errors and provide a common name basis for the remainder of the source.

Note:

There may only be one definition of a codeset and the definition must be the first item in the source file.

4.3.4 The DATA TABLES section

The DATA TABLES section must include the following tables:
- a property table
- a collation table
- a string table
- the two code conversion tables "toupper" and "tolower"

The property table
A property is the membership in a character class and can be accessed at runtime by the `ctype` library functions (described in "C-DS C V1.0 Programmer's Reference Manual" [6]).
A property table is introduced by the keyword PROPERTY. The default property table, built along with the codeset, has the predefined name PROP_DFLT. This is the table which will be accessed via the `setlocale()` function with the LC_CTYPE category.
A statement in the PROPERTY takes the following form:
Identifier':'*property_list*';'
where *Identifier* designates a defined code and *property_list* is a comma-separated list of known, user-defined properties.
Some properties also have effects on the interpretation of characters by various other international SINIX library routines.
The following properties are defined:

Property	**Key**
ILLEGAL	The corresponding code is defined but is not a legal code.
BLANK	The corresponding code is a blank.
CRTL	The corresponding code is a control code.
NUMERAL	The corresponding code is a number.
UPPER	The corresponding code is an uppercase letter.
LOWER	The corresponding code is a lowercase letter.
HEX	The corresponding (letter) code represents a hexadecimal digit.
PUNCT	The corresponding code is a punctuation mark.
SPACE	The corresponding code is a character that prints as space.
DIPHTHONG	The corresponding character is a diphthong. Diphthong for the purposes of the international SINIX database is defined as a character for which "one-to-two" collation must be used. (This implies an interdependence with the collation tables.)
MULTI	The corresponding character is constructed from a number of single-byte codes (at least two, no more than NL_NMAX) but is to be treated as a single code, i.e. an *n*-to-1 mapping. (Note: This allows two things: one is the expansion of 8-bit character sets to

	include multiletters (e.g. Ll, ll in Spanish) that collate *n* to one, and the other is the handling of 8/16-bit codes like ISO 6937). This property may only be used if the codeset is not a true 16-bit (EXTENDED) codeset.

Table 16: Defining properties in databases

The two most interesting properties in the list above are the properties DIPHTHONG and MULTI and their combinations for normal and extended codesets. To clarify the interdependence, here are some examples (both assuming an underlying 8-bit codeset with extensions, i.e. ISO 6937, and DIACRIT is a user-defined property):
For restrictions on property combination (a number must not be a blank, for example) refer to "X/Open Guide, System Interface Definitions" [13], pp. 44 - 48. `ic` checks the properties assigned to a character for mutual incompatibility.

Example

1. Definition of a multiletter
   Assuming the underlying codeset is ISO 6937, the German umlaut ä will be represented by two eight-bit codes: the nonspacing diacritical mark "diaresis" followed by the code for the character "a". If this "character" is to be collated "two to one" (for standard German dictionary order), the correct definition is:
   dia_a=diaresis,a:LOWER,DIACRIT,MULTI;
   which means that the code "dia_a" has the 16-bit value "c861" (diaresis==c8, a==61) and is a lowercase diacriticalled letter that collates "2 to 1".
2. Definition of a multidiphthong character
   Again taking the ISO 6937 codeset and the German umlaut ä, assume that this character is to be collated as the letter "a" followed by the letter "e" (German telephone book ordering). Now we have the case that a multiletter is to collate as two simple letters, essentially a "two-to-two" collation. The correct definition for this code now is:
   dia_a=diaresis,a:LOWER,DIACRIT,MULTI,DIPHTHONG;
   plus in the collation table section:
   dia=(a,e);

ISO 6937 is a typical example of an eight-bit codeset extended by a non-contiguous range of multiletters (not all combinations of diacritical marks and other codes are defined!).
It is legal to redefine the properties of a code. Only the definition in effect when the terminal symbols END '.' are reached will be put in the database.
A code with no defined property will be listed as ILLEGAL in the resultant property table.
A property table ends with the two terminal symbols END '.'.

The collation table

A collation table starts with the keyword COLLATION. The default collation table has the predefined named COLL_DFLT.
The LC_COLLATE category of the `setlocale()` function selects the corresponding collation table from a database.
The order of statements in the collation section is significant, as every statement starting with the keyword PRIMARY in the list of forms below opens up a new class of codes with the same primary weight and the primary weight (starting at 1) increases with each PRIMARY keyword encountered. In this way, for example, the fourth statement in the collation section assigns the primary weight 4 to all the codes named therein.

A statement in the collation section may take one of the following forms:

1. PRIMARY ':' *code_value_list*';'
2. PRIMARY ':' *Identifier* '-' *Identifier*';'
3. PRIMARY ':' REST';'
4. IGNORE ':' *code_value_list*';'
5. IGNORE ':' *Identifier* '-' *Identifier*';'
6. EQUAL ':' *Identifier* '-' *Identifier*';'
7. *Identifier* '=' '(' *Identifier* ',' *Identifier*')' ';'

The meaning of the first construction is to assign the codes designated in the *code_value_list* the same primary weight and ascending secondary weights from left to right. If the codes are to be sorted in order of codeset value, the second form of the collation statement may be used, assigning ascending secondary weights in ascending machine collation order.
The third construction (PRIMARY ':' REST';') will set the primary weight of all codes not explicitly named to the next applicable value (last primary weight assigned + 1), while the secondary weight will be assigned in ascending order of codeset value. This is a convenient notation for defaulting codes not explicitly specified to collate after or before all others.

Note:

When two strings are collated, the secondary weight is only significant when the strings have the same length. Otherwise only the primary weight is used. This means that if you want to collate numbers as in ASCII (1, 11, 12, 2, etc.), each of the numbers has to be given a unique primary weight.

If there are one or more codes which are not to be included by the REST notation, i.e. there are some codes which are to be IGNORED in the collation, the fourth or fifth type of statement can be used. These codes are not given any weights and will collate as if non-existent.
If a code is not given weights in the collation section, this code is treated as if it had the (otherwise illegal) primary and secondary weight zero. The net effect of this is that such a code will collate as if non-existent. For example, if '-' is such a code, then "abc" and "a-b-c" will collate the same.
The sixth construction (EQUAL ':' *Identifier* '-' *Identifier*';') can be used to assign a number of codes the same primary and secondary weights.
The seventh form of the collation statement (*Identifier* '=' '(' *Identifier* ',' *Identifier*')' ';') is reserved for the collation of diphthongs (one-to-two collation). The left-hand code collates as if it were the first right-hand code followed by the second right-hand code.

Note:

Please note that in order for diphthong collation to work correctly, the code named on the left-hand side of the statement must be marked as DIPHTHONG in at least one property table.

A collation table ends with the two terminal symbols END '.'.
Some examples of collations are given in section "Example of a database".

The string table

A string table starts with the keyword STRINGTABLE followed by the name of the string table. There is a default string table which is internally named STRG_DFLT.
Each statement in a string table is in the form:

Identifier '=' *format_value* ';'
where *Identifier* is an identifier, i.e., the name of the string, and *format_value* is a comma-separated list of strings, character constants and *Identifiers* designating codes. This allows you to incluude non-ASCII codes in any string table value by giving the name of the code in *format_value*.

Example
The expression
MON_3="M", dia_a, "rz";
stands for *März*, which is the German for March.
The compiler will check the existence of all codes in the *format_value* list, compiling escaped characters in string constants in the process.
If the advice of using header files in the codeset section was followed and the names for characters are the same in the header files, the string table can be copied (or again included) in several sources.
The string table must contain at least the following strings (the order in which they are listed is not significant):

Parameter (identifier)	**Corresponding category**	**Description**
D_FMT	LC_TIME	Default date format
T_FMT	LC_TIME	Default time format
D_T_FMT	LC_TIME	Default date and time format
AM_STR	LC_TIME	English AM string (midnight to midday)
PM_STR	LC_TIME	English PM string (midday to midnight)
DAY_*x*	LC_TIME	Day of week (1 to 7)
ABDAY_*x*	LC_TIME	Abbreviated weekday name
MON_*x*	LC_TIME	Month (1 to 12)
ABMON_*x*	LC_TIME	Abbreviated month name
DFLT_DATE	LC_TIME	Default date format for `date` command
TIME_FMT	LC_TIME	Default time format without seconds
DAY_FMT	LC_TIME	Default short date format without year
FULL_DAY	LC_TIME	Default date format without year
AR_DATE	LC_TIME	Default date format for `ar(1)`
LAST_DATE	LC_TIME	Default date format for `last(1)`
LS_DATE	LC_TIME	Default date format for `ls(1)`, `who(1)`
LS_DATE2	LC_TIME	Default date format for `ls(1)` > 6 months

PS_DATE	LC_TIME	Default date format for `ps(1)`
SU_DATE	LC_TIME	Default date format for `su(1)`
TAR_DATE	LC_TIME	Default date format for `tar(1)`, `cpio(1)`, `pr(1)`
YESSTR	LC_MESSAGES	yes string
NOSTR	LC_MESSAGES	no string
QUITSTR	LC_MESSAGES	Quit string
CRNCYSTR	LC_MONETARY	Currency symbol
RADIXCHAR	LC_NUMERIC	Radix character
THOUSEP	LC_NUMERIC	Digit grouping symbol

Table 17: Mandatory strings by category

Parameter (identifier)	Corresponding category	Description
T_FMT_AMPM	LC_TIME	Time format on 12-hour clock with AM_PM
ALT_DIGITS	LC_TIME	Alternate digits symbols
ERA	LC_TIME	Era definition
ERA_D_FMT	LC_TIME	Default era date format
ERA_T_FMT	LC_TIME	Default era time format
ERA_D_T_FMT	LC_TIME	Default era date and time format

Table 18: Optional strings by category

The following string are optional
Depending on the associated category, the information is loaded from the string table by a call to the `setlocale` function, i.e. `setlocale`(*category*, *locale*) or `setlocale`(`LC_ALL`, *locale*). A string table ends with the two terminal symbols END '.'.

The conversion tables
In an environment where more than one codeset is supported, the ability to convert from one codeset to another becomes of prime importance. Characters can be converted with the `iconv` command, which is described in the manual "SINIX V5.41 Commands, Volume 1" [1].

There is also one type of possible conversion within a codeset. This type of conversion will hereafter be called a code conversion. Examples are the standard conversions `toupper` and `tolower`.
A conversion table starts with the keyword CONVERSION. Each conversion must be given a name with which it will be accessed at runtime. The two code conversions `toupper` and `tolower` are mandatory. Names of conversions must be unique throughout the source file.
A statement in a conversion table takes one of the following forms:
Identifier '->' *conversion_value* ';'
Identifier '-' *Identifier* '->' *Identifier* '-' *Identifier* ';'
DEFAULT '->'*default_value*';'
where *Identifier* specifies a code defined in the codeset and *conversion_value* the code value the left-hand side should be converted to.
The *default_value* for a conversion may be given using the DEFAULT statement. Any code for which there is no explicit conversion specified will map to the given value. There are two predefined values possible in a DEFAULT statement, namely VOID and SAME.
VOID means that all other codes will convert either to ASCII `NUL` (in the case of a code conversion) or a null string (in the case of a string conversion).
SAME, which means that a code is converted to itself if there is no explicit conversion given in the section.
Identifier '-'*Identifier* denotes a range and implies an underlying machine collation sequence. This notation is only valid for code conversions where such a collation sequence is always defined.
If no DEFAULT clause is given, the default clause is assumed to read
DEFAULT '- >' VOID';'
A conversion table ends with the two terminal symbols END '.'.
Some examples of both types of conversions are given in section "Syntax description".

4.3.5 Syntax description

This section describes an EBNF notation of the syntax recognized by the *ic* compiler. All terminal symbols start with capital letters or are enclosed in single quotes. Keywords are capitalized throughout.
The following rules apply to *Identifier* names:
- An identifier must begin with an underscore '_' or a letter.
- A string of letters, digits or the special characters '_' and '.' of any length may be used after the first character.
- If an identifier contains the special character '.', this character must be followed by at least one letter, digit or '_' (i.e. '.' may not be the last character of an identifier name).

The following rules apply to *String* names:
- A string must be enclosed in double quotes.
- A string may not extend over several lines.
- A string may contain the following C-like escape sequences:

\n	ASCII newline
\r	ASCII return
\t	ASCII tabulator

\b	ASCII backspace
\f	ASCII formfeed
\\	escaped backslash
\"	escaped double quotes

Table 19: Escape sequences in databases

- A string may not be longer than 255 characters.

A *Constant* may take one of the following forms:

- A character constant, i.e. a character enclosed in single quotes. The C rules for escaping characters in character constants apply.
- A hexadecimal constant of the form 0x*nnnn* where '*n*' is a hexadecimal digit in the range [0-9, A-F, a-f]. Leading zeros need not be specified. The value must be representable by at most 16 bits.
- An octal constant of the form 0*nnnn* where '*n*' is an octal digit in the range [0-7]. Leading zeros need not be specified. The value must be representable by at most 16 bits.
- A character in ISO notation *n*/*n* where '*n*' is a decimal number in the range [0-15].
- A decimal number *n* where '*n*' is a positive integer which can be represented with 16 bits.
- Blanks (' ', newline, tabulator) are considered character separators.

C-like comments starting with /* and ending with */ may be inserted at any position in which blanks are allowed. Comments always terminate a token.

As the source is first processed using the C preprocessor, ordinary preprocessor directives, such as '#include', '#define', '#if' etc., may appear in the source.

Grammar definition

```
intl_data_base
        : codeset_table mandatory_tables data_tables
mandatory_tables
        : property_table collation_table format_table conversion_table
        | collation_table format_table conversion_table
data_tables
        : data_table
        | data_tables data_table
data_table
        : property_table
        | collation_table
        | format_table
        | conversion_table
        |
codeset_table
        : code_set_intro code_definition_list end
code_set_intro
        : CODESET Identifier ':'
        | EXTENDED CODESET Identifier ':'
end     : END'.'
        | ';' END'.'
code_definition_list
```

```
        : code_definition
        | code_definition_list ';' code_definition
code_definition
        : code_def_intro ':' property_list
        | code_def_intro
        | property_definition
code_def_intro
        : Identifier '=' value_list
value_list
        : code
        | value_list ',' code
code  : Constant
        | Identifier
property_list
        : Property
        | property_list ',' Property
property_table
        : PROPERTY Identifier ':' property_definition_list end
property_definition_list
        : property_definition
        | property_definition_list ';' property_definition
property_definition
        : Identifier ':' property_list
        | DEFAULT ':' Identifier
collation_table
        :COLLATION ':' collation_list end
collation_list
        : collation
        | collation_list ';'collation
collation
        : collation_head code_value_list
        | collation_head Identifier '-' Identifier
        | collation_head REST
        | Identifier '=' '(' Identifier ',' Identifier ')'
        | IGNORE ':' code_ignore_list
        | IGNORE ':' Identifier '-' Identifier
collation_head
        : PRIMARY ':'
        | EQUAL ':'
code_value_list
        : Identifier
        | code_value_list ',' Identifier
code_ignore_list
        : Identifier
        | code_ignore_list ',' Identifier
format_table
        : STRINGTABLE ':' format_list end
format_list
```

```
        : format
        | format_list ';' format
format
        : Identifier '=' format_value
format_value
        : code_or_string
        | format_value ',' code_or_string
code_or_string
        : code
        | String
conversion_table
        : conversion_head conversion_list end
conversion_head
        : CONVERSION Identifier ':'
conversion_list
        : conversion
        | conversion_list ';' conversion
conversion
        : DEFAULT IS default_value
        | Identifier IS conversion_value
        | Identifier '-' Identifier IS Identifier '-' Identifier
default_value
        : VOID
        | SAME
        | conversion_value
conversion_value
        : code
        | conversion_value ',' code
```

4.3.6 Example of a database

This section consists of an example in the form of a complete database or locale named De_DE.88591.

```
/*
* This is a complete example of the German database for ISO 8859-1.
* Assuming a source file called De.88591.in to be compiled by:
* ic -v De.88591.in
*
* ic will write the following diagnostic on standard error:
*
* international database De_DE.88591:
* 256 code table entries (256 simple/0 multi-byte).
* 1 property table(s).
* 1 collation table(s).
* 1 string table(s).
* 3 conversion tables: canon, tolower, toupper.
* 9481 bytes total length.
*
* and a directory named De_DE.88591 (containing all informations to
* produce and install the De_DE.88591 locale) will be created in
```

```
* current directory.
*
* Note: ic uses the standard C pre-processor (usually /lib/cpp) so it
* has to be present on your machine.
*/
/*
* Next an eight bit codeset is defined and the compiled database
* given the name De_DE.88591
*/
CODESET De_DE.88591 :
/*
* Now the codeset has to be defined in the form:
* Identifier =value_list [ : Properties ] ;
* where value may be given as C-style character constant, in octal,
* decimal or ISO notation, or by giving the name of a previously
* defined code.
* Remember that the list of simple codes must contain ALL code
* value from 0x0 up to and including the code with the highest
* value defined, in our case 0xff.
*/
nul = 0x00 : CTRL;        soh= 0x01 : CTRL;
stx = 0x02 : CTRL;        etx= 0x03 : CTRL;
eot = 0x04 : CTRL;        enq= 0x05 : CTRL;
ack = 0x06 : CTRL;        bel= 0x07 : CTRL;
bs = 0x08 : CTRL;         tab= 0x09 : SPACE,CTRL;
lf = 0x0a : SPACE, CTRL;  vt= 0x0b : SPACE,CTRL;
ff = 0x0c : SPACE, CTRL;  cr= 0x0d : SPACE,CTRL;
so = 0x0e : CTRL;         si= 0x0f : CTRL;
dle = 0x10 : CTRL;        dc1= 0x11 : CTRL;
dc2 = 0x12 : CTRL;        dc3= 0x13 : CTRL;
dc4 = 0x14 : CTRL;        nak= 0x15 : CTRL;
syn = 0x16 : CTRL;        etb= 0x17 : CTRL;
can = 0x18 : CTRL;        em= 0x19 : CTRL;
sub = 0x1a : CTRL;        esc= 0x1b : CTRL;
fs = 0x1c : CTRL;         gs= 0x1d : CTRL;
rs = 0x1e : CTRL;         us= 0x1f : CTRL;
space = ' '               : SPACE, BLANK;
exclmark = '!'            : PUNCT ;
quotes = '"'              : PUNCT ;
percent = '%'             : MISCEL ;
at = '&'                  : MISCEL ;
apostr = 2/7              : PUNCT ;
leftpar = '('             : PUNCT ;
rightpar = ')'            : PUNCT ;
asterisk = '*'            : MISCEL ;
plus = '+'        : ARITH ;
comma = ','       : PUNCT ;
minus = '-'       : ARITH ;
dot = '.'         : PUNCT ;
solidus = '/'     : PUNCT ;
zero = '0'        : NUMERAL, HEX ;
one = '1'         : NUMERAL, HEX ;
two = '2'         : NUMERAL, HEX ;
three = '3'       : NUMERAL, HEX ;
```

```
four = '4'          : NUMERAL, HEX ;
five = '5'          : NUMERAL, HEX ;
six = '6'           : NUMERAL, HEX ;
seven = '7'         : NUMERAL, HEX ;
eight = '8'         : NUMERAL, HEX ;
nine = '9'          : NUMERAL, HEX ;
colon = ':'         : PUNCT ;
semi = ';'          : PUNCT ;
less = '<'          : ARITH ;
equal = '='         : ARITH ;
greater = '>'       : ARITH ;
questmark = '?'     : PUNCT ;
A = 'A'             : UPPER, HEX ;
B = 'B'             : UPPER, HEX ;
C = 'C'             : UPPER, HEX ;
D = 'D'             : UPPER, HEX ;
E = 'E'             : UPPER, HEX ;
F = 'F'             : UPPER, HEX ;
G = 'G'             : UPPER ;
H = 'H'             : UPPER ;
I = 'I'             : UPPER ;
J = 'J'             : UPPER ;
K = 'K'             : UPPER ;
L = 'L'             : UPPER ;
M = 'M'             : UPPER ;
N = 'N'             : UPPER ;
O = 'O'             : UPPER ;
P = 'P'             : UPPER ;
Q = 'Q'             : UPPER ;
R = 'R'             : UPPER ;
S = 'S'             : UPPER ;
T = 'T'             : UPPER ;
U = 'U'             : UPPER ;
V = 'V'             : UPPER ;
W = 'W'             : UPPER ;
X = 'X'             : UPPER ;
Y = 'Y'             : UPPER ;
Z = 'Z'             : UPPER ;
low_line = '_'      : PUNCT ;
a = 'a'             : LOWER, HEX ;
b = 'b'             : LOWER, HEX ;
c = 'c'             : LOWER, HEX ;
d = 'd'             : LOWER, HEX ;
e = 'e'             : LOWER, HEX ;
f = 'f'             : LOWER, HEX ;
g = 'g'             : LOWER ;
h = 'h'             : LOWER ;
i = 'i'             : LOWER ;
j = 'j'             : LOWER ;
k = 'k'             : LOWER ;
l = 'l'             : LOWER ;
m = 'm'             : LOWER ;
n = 'n'             : LOWER ;
o = 'o'             : LOWER ;
```

```
p = 'p'           : LOWER ;
q = 'q'           : LOWER ;
r = 'r'           : LOWER ;
s = 's'           : LOWER ;
t = 't'           : LOWER ;
u = 'u'           : LOWER ;
v = 'v'           : LOWER ;
w = 'w'           : LOWER ;
x = 'x'           : LOWER ;
y = 'y'           : LOWER ;
z = 'z'           : LOWER ;
del = 0x7f        : CTRL;
numbsign = '#'    : MISCEL ;
dollar = '$'      : CURRENCY ;
commt = '@'       : MISCEL ;
l_sq_br = '['     : MISCEL ;
revsolidus = ''   : MISCEL ;
r_sq_br = ']'     : MISCEL ;
circ = '^'        : MISCEL ;
gravcc = '''      : MISCEL ;
l_cur_br = '{'    : MISCEL ;
ver_line = ''     : MISCEL ;
r_cur_br = '}'    : MISCEL ;
tilde = '~'       : MISCEL ;
sc00 = 0x80; sc01 = 0x81; sc02 = 0x82; sc03 = 0x83;
sc04 = 0x84; sc05 = 0x85; sc06 = 0x86; sc07 = 0x87;
sc08 = 0x88; sc09 = 0x89; sc0a = 0x8a; sc0b = 0x8b;
sc0c = 0x8c; sc0d = 0x8d; sc0e = 0x8e; sc0f = 0x8f;
sc10 = 0x90; sc11 = 0x91; sc12 = 0x92; sc13 = 0x93;
sc14 = 0x94; sc15 = 0x95; sc16 = 0x96; sc17 = 0x97;
sc18 = 0x98; sc19 = 0x99; sc1a = 0x9a; sc1b = 0x9b;
sc1c = 0x9c; sc1d = 0x9d; sc1e = 0x9e; sc1f = 0x9f;
nbsp = 10/0        : SPACE;
revexcl = 10/1     : PUNCT ;
cent = 10/2        : CURRENCY ;
pound = 10/3       : CURRENCY ;
currency = 10/4    : CURRENCY ;
yen = 10/5         : CURRENCY ;
vertbar = 10/6     : MISCEL ;
section = 10/7     : MISCEL ;
diaresis = 10/8    : MISCEL ;
copyright = 10/9   : MISCEL ;
fem_ord = 10/10    : MISCEL ;
ang_q_l = 10/11    : PUNCT ;
not = 10/12        : MISCEL ;
shy = 10/13        : MISCEL ;
register = 10/14   : MISCEL ;
macron = 10/15     : MISCEL ;
degree = 11/0      : MISCEL ;
plu_min = 11/1     : ARITH ;
sup_two = 11/2     : SUPSUB ;
sup_three = 11/3   : SUPSUB ;
acute = 11/4       : MISCEL ;
micro = 11/5       : MISCEL ;
```

```
pilcrow = 11/6      : MISCEL ;
mid_dot = 11/7      : MISCEL ;
cedilla = 11/8      : MISCEL ;
sup_one = 11/9      : SUPSUB ;
mas_ord = 11/10     : MISCEL ;
ang_q_r = 11/11     : PUNCT ;
o_quart = 11/12     : FRACTION ;
half = 11/13        : FRACTION ;
t_quart= 11/14      : FRACTION ;
revquest = 11/1     : PUNCT ;
GRAVE_A = 12/0      : UPPER ;
ACUTE_A = 12/1      : UPPER ;
CIRC_A = 12/2       : UPPER ;
TILDE_A = 12/3      : UPPER ;
DIA_A = 12/4        : UPPER ;
RING = 12/5         : UPPER ;
AE = 12/6           : UPPER ;
CEDIL_C = 12/7      : UPPER ;
ACUTE_E = 12/9      : UPPER ;
CIRC_E = 12/10      : UPPER ;
DIA_E = 12/11       : UPPER ;
GRAVE_I = 12/12     : UPPER ;
ACUTE_I = 12/13     : UPPER ;
CIRC_I = 12/14      : UPPER ;
DIA_I = 12/15       : UPPER ;
ETH = 13/0          : UPPER ;
TILDE_N = 13/1      : UPPER ;
GRAVE_O = 13/2      : UPPER ;
ACUTE_O = 13/3      : UPPER ;
CIRC_O = 13/4       : UPPER ;
TILDE_O = 13/5      : UPPER ;
DIA_O = 13/6        : UPPER ;
multiply = 13/7     : ARITH ;
SLASH_O = 13/8      : UPPER ;
GRAVE_U = 13/9      : UPPER ;
ACUTE_U = 13/10     : UPPER ;
CIRC_U = 13/11      : UPPER ;
DIA_U = 13/12       : UPPER ;
ACUTE_Y = 13/13     : UPPER ;
THORN = 13/14       : UPPER ;
sharp_s = 13/15     : LOWER ;
grave_a = 14/0      : LOWER ;
acute_a = 14/1      : LOWER ;
circ_a = 14/2       : LOWER ;
tilde_a = 14/3      : LOWER ;
dia_a = 14/4        : LOWER ;
ring_a = 14/5       : LOWER ;
ae = 14/6           : LOWER ;
cedil_e = 14/7      : LOWER ;
grave_e = 14/8      : LOWER ;
acute_e = 14/9      : LOWER ;
circ_e = 14/10      : LOWER ;
dia_e = 14/11       : LOWER ;
grave_i = 14/1      : LOWER ;
```

```
acute_i = 14/13     : LOWER ;
circ_i = 14/14      : LOWER ;
dia_i =14/15        : LOWER ;
eth = 15/0          : LOWER ;
tilde_n = 15/1      : LOWER ;
grave_o = 15/2      : LOWER ;
acute_o = 15/3      : LOWER ;
circ_o = 15/4       : LOWER ;
tilde_o = 15/5      : LOWER ;
dia_o = 15/6        : LOWER ;
divide = 15/7       : ARITH ;
slash_o = 15/8      : LOWER ;
grave_u = 15/9      : LOWER ;
acute_u = 15/10     : LOWER ;
circ_u = 15/11      : LOWER ;
dia_u = 15/12       : LOWER ;
acute_u = 15/13     : LOWER ;
thorn = 15/14       : LOWER ;
dia_y = 15/15       : LOWER ;
/*
* The defined codeset is equal for all languages supporting ISO 8859/1
* so if one wants to define databases for more languages one should
* write the codeset to a file to include it in the different data-
* base sources (with the use of the #include facility).
* A default property table "PROP_DFLT" is created together with the
* codeset table. As already mentioned some properties (i.e., DIPHTHONG
* and MULTI) have special meaning to the collation. If the form
* "PROPERTY : Property_table_name;" is not used in the collation
* table the default property table "PROP_DFLT" is used as base for the
* collation. From defined codeset it is to be seen that the German
* sharp_s is already defined as:
*         sharp_s = 13/15 : LOWER ;
* In German dictionary sharp_s should be sorted as "ss". This means
* it has to have the property DIPHTHONG defined. In the codeset section
* one has the possibility to redefine properties as done here for sharp_s.
* The last assignment is significant and saved in the database.
*/
sharp_s          : LOWER, DIPHTHONG ;
END.
/*
* Exactly one nameless collation table must exist in the database source.
* Internally it is given the name "COLL_DFLT".
* The order of statements in the collation section is significant.
* Every time the keyword PRIMARY is used a new primary weight is
* assigned. If PRIMARY is followed by a comma separated list of
* codes (see "PRIMARY : a, dia_a;" below) or the REST notation
* is used, the codes are given the same primary weight and
* ascending secondary weight.
* NOTE: The secondary weight is only significant when the strings
* collated have the same length. If numbers should collate
* as in ASCII (1, 11, 12, 2, 3) each number has to be given
* a unique primary weight.
*/
COLLATION :
```

```
PRIMARY: nul ;          PRIMARY: soh ;          PRIMARY: stx ;          PRIMARY: etx ;
PRIMARY: eot ;          PRIMARY: enq ;          PRIMARY: ack ;          PRIMARY: be l ;
PRIMARY: bs ;           PRIMARY: tab ;          PRIMARY: lf ;           PRIMARY: vt ;
PRIMARY: ff ;           PRIMARY: cr ;           PRIMARY: so ;           PRIMARY: si ;
PRIMARY: dle ;          PRIMARY: dc1 ;          PRIMARY: dc2 ;          PRIMARY: dc3 ;
PRIMARY: dc4 ;          PRIMARY: nak ;          PRIMARY: syn ;          PRIMARY: etb ;
PRIMARY: can ;          PRIMARY: em ;           PRIMARY: sub ;          PRIMARY: esc ;
PRIMARY: fs ;           PRIMARY: gs ;           PRIMARY: rs ;           PRIMARY: us ;
PRIMARY: space ;        PRIMARY: exclmark ;
PRIMARY: quotes ;       PRIMARY: numbsign ;
PRIMARY: dollar ;       PRIMARY: percent ;
PRIMARY: at ;           PRIMARY: apostr ;
PRIMARY: leftpar ;      PRIMARY: rightpar ;
PRIMARY: asterisk ;     PRIMARY: plus ;
PRIMARY: comma ;        PRIMARY: minus ;
PRIMARY: dot ;          PRIMARY: solidus ;
PRIMARY: zero ;         PRIMARY: one ;          PRIMARY: two ;
PRIMARY: three ;        PRIMARY: four ;         PRIMARY: five ;
PRIMARY: six ;          PRIMARY: seven;         PRIMARY: eight ;
PRIMARY: nine ;
PRIMARY: colon ;        PRIMARY: semi ;         PRIMARY: less ;
PRIMARY: equal ;        PRIMARY: greater ;      PRIMARY: questmark ;
PRIMARY: section ;
PRIMARY: a, dia_a;
PRIMARY: b;             PRIMARY: c;             PRIMARY: d;
PRIMARY: e;             PRIMARY: f;             PRIMARY: g;
PRIMARY: h;             PRIMARY: i;             PRIMARY: j;
PRIMARY: k;             PRIMARY: l;             PRIMARY: m;
PRIMARY: n;
PRIMARY: o, dia_o;
PRIMARY: p;             PRIMARY: q;             PRIMARY: r;
PRIMARY: s;             PRIMARY: t;
PRIMARY: u, dia_u;
PRIMARY: v;             PRIMARY: w;             PRIMARY: x;
PRIMARY: y;             PRIMARY: z;
PRIMARY: circ ;         PRIMARY: low_line;      PRIMARY: grav_acc;
PRIMARY: A, DIA_A;
PRIMARY: B;             PRIMARY: C;             PRIMARY: D;
PRIMARY: E;             PRIMARY: F;             PRIMARY: G;
PRIMARY: H;             PRIMARY: I;             PRIMARY: J;
PRIMARY: K;             PRIMARY: L;             PRIMARY: M;
PRIMARY: N;
PRIMARY: O, DIA_O;
PRIMARY: P;             PRIMARY: Q;             PRIMARY: R;
PRIMARY: S;             PRIMARY: T;
PRIMARY: U, DIA_U;
PRIMARY: V;             PRIMARY: W;             PRIMARY: X;
PRIMARY: Y;             PRIMARY: Z;
sharp_s = (s,s);
PRIMARY: del;
PRIMARY: REST;
END.
/*
* Exactly one nameless string table must exist in the database source.
```

```
* Internally it is given the name "STRG_DFLT".
* The stringtable has to contain a set of monetary strings as described
* in ic( ) and langinfo( ). Ic will complain if any of the the monetary
* strings are missing. In addition any number of strings can be defined.
*/
STRINGTABLE:
RADIXCHAR     = comma;
THOUSEP       = dot;
YESSTR        = "ja";
NOSTR         = "nein";
CRNCYSTR      = "DM";
D_T_FMT       = "%a %d.%h.19%y, %T %Z";
D_FMT         = "%m.%d.%y" ;
T_FMT         = "%T%Z";
AM_STR        = "AM";
PM_STR        = "PM" ;
DFLT_DATE     = "%a %e.%b.%Y, %T %Z" ;
TIME_FMT      = "%H:%M" ;
DAY_FMT       = "%d.%m" ;
FULL_DAY      = "%a %e.%b" ;
AR_DATE       = "%b %d %H:%M %Y" ;
LAST_DATE     = "%a %e.%b %H:%M" ;
LS_DATE       = "%h %d %H:%M" ;
LS_DATE2      = "%h %d 19%y" ;
PS_DATE       = "%d.%b" ;
SU_DATE       = "%d.%m %H:%M" ;
TAR_DATE      = "%e.%b %H:%M %Y" ;
QUITSTR       = "abbrechen" ;
T_FMT_AMPM    = "%I:%M:%S %p";
DAY_1         = "Sonntag";     DAY_2         = "Montag";
DAY_3         = "Dienstag";    DAY_4         = "Mittwoch";
DAY_5         = "Donnerstag";  DAY_6         = "Freitag";
DAY_7         = "Samstag";
ABDAY_1       = "So";          ABDAY_2       = "Mo";
ABDAY_3       = "Di";          ABDAY_4       = "Mi";
ABDAY_5       = "Do";          ABDAY_6       = "Fr";
ABDAY_7       = "Sa";
MON_1         = "Januar";      MON_2         = "Februar";
MON_3         = M, dia_a,      MON_4         = "April";
MON_5         = "Mai";         MON_6         = "Juni";
MON_7         = "Juli";        MON_8         = "August";
MON_9         = "September";   MON_10        = "Oktober";
MON_11        = "November";    MON_12        = "Dezember";
ABMON_1       = "Jan";         ABMON_2       = "Feb";
ABMON_3       = M, dia_a, r;   ABMON_4       = "Apr";
ABMON_5       = "Mai";         ABMON_6       = "Jun";
ABMON_7       = "Jul";         ABMON_8       = "Aug";
ABMON_9       = "Sep";         ABMON_10      = "Okt";
ABMON_11      = "Nov";         ABMON_12      = "Dez";
END.
/*
* Now only the two mandatory conversion tables "toupper" and
* "tolower" are missing. This is a conversion within the codeset
* itself so the keyword CONVERSION should be used.
```

```
*/
CONVERSION tolower :
DEFAULT      -> SAME;
A - Z        -> a - z;
DIA_A        -> dia_a;
DIA_O        -> dia_o;
DIA_U        -> dia_u;
END.
CONVERSION toupper :
DEFAULT      -> SAME;
a - z        -> A - Z;
dia_a        -> DIA_A;
dia_o        -> DIA_O;
dia_u        -> DIA_U;
END.
END.
```

4.4 Generating message catalogs

Messaging systems allow program messages to be held apart from the logic of the program, translated into other languages and retrieved at runtime according to the language requirements of each user.
A messaging system is based on a set of library functions. Calls to these functions replace hard-coded messages in the program text, so that the functions retrieve the messages from the message catalog appropriate to the locale applicable at program runtime.
The SINIX V5.41 operating system supplies two messaging systems:
- the X/Open messaging interface
- the AT&T System V messaging interface

As the X/Open messaging system is a far more flexible interface than the AT&T system, only the X/Open messaging interface is described at this point. There is a brief description of the AT&T messaging system in section "Appendix C: Message catalogs as in AT&T System V, Rel. 4.0".
Both messaging mechanisms provide all the essential program internationalization functionality. However, since the two systems work in very different ways, it is advisable to use only one messaging system in any given application.

4.4.1 X/Open-compliant message catalogs

As mentioned in section "Message catalogs (LC_MESSAGES category)", the X/Open messaging system on SINIX V5.41 systems offers the additional convenience that you can use the NLSPATH variable to define the message catalog search path. The NLSPATH variable is evaluated by the `catopen()` function. This allows you to store message catalogs anywhere in the file system and access them without having to recompile the application. NLSPATH essentially uses the same conventions as the PATH environment variable, but `catopen()` also evaluates certain parts of the value of the LANG environment variable, provided that LANG is in X/Open *language_territory.codeset* syntax.
The following metacharacters are interpreted:

Metachar acter	**Key**
%L	expands to the value of LANG
%l	expands to the *language* part of the value of LANG
%t	expands to the *territory* part of the value of LANG
%c	expands to the *codeset* part of the value ofLANG
%N	expands to the name of the message catalog
%%	stands for a percent sign '%'

Table 20: Metacharacters in the NLSPATH environment variable

If the NLSPATH environment variable is not set, its value defaults to:
`NLSPATH=/usr/lib/nls/msg/%l/%N.cat:./%N.cat`
This means that the system looks for message catalogs first in `/usr/lib/nls/msg/%l/`, then in the current working directory.

4.4.2 Commands

gencat
produces a binary-encoded message catalog from one or more message text source files.
dumpmsg
produces a human-readable message text source file from a binary-encoded message catalog. This is useful if, for example, you only have the binary message catalog and need to translate the messages into another language or make corrections to the existing messages.
iecho
writes a message from a message catalog to standard output.
extract
suitable for internationalizing the message texts of an existing program.
For full details of these commands refer to the manuals "SINIX V5.41 Commands, Volumes 1-3" [1-3].

4.4.3 C library functions

catopen ( )
locates a named message catalog in the file store and opens it for use by `catgets()` and `catclose()`.
catgets ( )
retrieves messages from a message catalog opened by `catopen()`.
catclose ( )
closes a message catalog.
For full details of these functions refer to the manual "C-DS C V1.0 Programmer's Reference Manual" [6].

4.4.4 Message text source file format

To be processed by `gencat`, message text source files must be in the following format:
$quote "
$set 1
1 " message_text "
2 " message_text "
The fields in each line must be separated by one space character. If there is no `$quote` directive, any extra space characters are viewed as belonging to the next field.
$set n comment
This line specifies the set identifier for the following messages until the next *$set*, *$delset* or end-of-file appears. *n* denotes the set identifier and must be in the range [1,{NL_SETMAX}]. Within a catalog the set identifiers must be in ascending order but need not be contiguous. Any string following the set identifier is treated as a comment. If no *$set* directive is defined, all messages will be assigned a default set identifier NL_SETD. The value of NL_SETD is defined in the *nl_types.h* header file.
$delset n comment
This line deletes message set *n* from an existing message catalog. *n* denotes the set identifier. Any string following the set identifier is treated as a comment.
$ comment
A line beginning with *$* followed by whitespace is treated as a comment.
m message_text

m denotes the message identifier, which must be in the range [1,{NL_MSGMAX}]. The *message_text* is stored in the message catalog with the set identifier specified by the last *$set* directive and with message identifier *m*. If *message_text* is empty and the message identifier is followed by a blank or a tab, an empty string is stored in the message catalog; but if there is no blank or tab following the message identifier and *message_text* is omitted, the message with the specified identifier is deleted from the catalog. Within a catalog the set identifiers must be in ascending order but need not be contiguous. The length of *message_text* must be in the range [0,{NL_TEXTMAX}].

$quote c

This line specifies an optional quote character *c*, which can be used to surround the message text so that leading and trailing spaces and empty message texts are visible in a message source file.

Empty lines in a message text source file are ignored. Message texts may contain the following special characters and escape sequences:

Description	Symbol	Sequence
newline	NL(LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
formfeed	FF	\f
backslash	\	\\
octal bit pattern	ddd	\ddd
hex bitpattern	0xnn	\0xnn

Table 21: Special characters and escape sequences in message texts

The escape sequence *ddd* consists of a backslash followed by one, two or three octal digits specifying the value of the required character.

The escape sequence *0xnn* consists of a backslash followed by *0x* and one or two hexadecimal digits specifying the value of the required character.

A backslash followed by a newline character is also used to continue a string on the following line. Thus the following two lines define a single message string:

```
1 This line \
continues on the next line.
```

which is equivalent to:

```
1 This line continues on the next line.
```

4.4.5 Sample message text source file

```
$quote "
$set 1
1 " This is message 1"
2 " This is message 2"
5 " This is message 5"
$set 3
3 " This is message 3"
```

```
4 " This is message 4"
```

4.4.6 Accessing message catalogs

Message catalogs are opened ready for use by a call to the `catopen()` library function, which locates the identified message catalog according to the search and naming rules defined in the NLSPATH environment variable.

Example

`nl_catd=catopen("`*`catalog_name`*`",0);`
`catopen()` returns a catalog descriptor `nl_catd`, which is used on subsequent calls to the `catgets()` function to identify the prepared message catalog. Message catalogs are closed by a call to the `catclose()` library function.

5 Using internationalized software

The functions discussed in previous chapters primarily relate to developing internationalized software and to localizing this software by creating locale-specific databases. This chapter deals with using internationalized software and localized databases on operating system level.
The runtime behavior of an internationalized software product is governed by the locale from which it takes its country-specific information. System administrators and users can set environment variables to specify access to a particular locale or just parts of it, thereby setting up customized working environments.

5.1 Guided tour of the system

The standard SINIX V5.41 operating system package includes a package named "SINIX NLS databases" `(SInlsdb)` which supplies German and English locales/databases created by Siemens Nixdorf Informationssysteme AG.
For information on other databases consult your local Siemens Nixdorf office.
If you create your own databases you must comply with set naming conventions based on the conventions defined by X/Open. Databases and locales that you create must also be stored in defined directories on the system. The following sections deal with the SINIX naming conventions and identify the standard directories for user-defined databases.

5.1.1 Locale naming conventions

As there are some UNIX systems where file names are restricted to a maximum of 14 characters, locale names have to be limited to 14 characters to guarantee portability. By default the name of a locale is made up of the components *language_territory.codeset*.
Each of these components supplies the environment with significant information of various types. The SINIX V5.41 operating system assumes that the name of a locale is in the notation *Ll_TT.codeset*.
This means that the name complies with the syntax defined by X/Open (see "X/Open Guide, System Interface Definitions" [13]).
The *Ll* element stands for *language* and refers to the native language the system is to support. The ISO 639 Standard among other things defines a set of standardized language name codes for use in technical documentation and related fields.
The following table is extracted from the ISO 639 standard.

Code	Language	Code	Language
Da	Danish	De	German
En	English	Es	Spanish
Fi	Finnish	Fr	French
El	Greek	It	Italian
Nl	Dutch	No	Norwegian
Pt	Portuguese	Sv	Swedish

Tr	Turkish		

Table 22: ISO 639 Standard language name codes

The *TT* element stands for *territory* and generally refers to a particular country; but it may also identify a group of countries or a region within one country.
The ISO IS3166 Standard defines a set of standardized country codes. The following table is an extract from the standard.

Code	**Country**	**Code**	**Country**
AT	Austria	BE	Belgium
CA	Canada	CH	Switzerland
DK	Denmark	DE	Germany
FI	Finland	FR	France
GB	Great Britain	GR	Greece
IE	Ireland	IT	Italy
LI	Liechtenstein	LU	Luxembourg
NL	Holland	NO	Norway
PT	Portugal	SP	Spain
SE	Sweden	TR	Turkey
US	United States		

Table 23: ISO IS3166 Standard country codes

The *codeset* element stands for the codeset to be used. As codeset identifiers can be fairly long, they should be reduced to their essentials when used as part of a locale name. The *codeset* may, for example, take the following form:
`646`equivalent toISO 646 (US-ASCII)
`88591`equivalent toISO 8859-1 (ISO Latin-1)
`88595`equivalent toISO 8859-5 (ISO Latin-Cyrillic)

Note:

One exception to this naming convention is the "C" locale, which designates a minimal locale available on every system. The "C" locale is based on the ASCII character set and American English. If a program does not set a locale, it uses "C" by default.

5.1.2 Storing locale-specific information

Most locale-specific information resides in the directory
`/usr/lib/locale`
and its subdirectories. Apart from the `/usr/lib/locale/TZ` directory, which simply contains timezone information, all the subdirectories of `/usr/lib/locale` are named after the locales available on the system. If the databases of the *SINIX NLS databases* package (`SInlsdb`) are installed, they will also be stored in `/usr/lib/locale` in the form described below.
Every fully defined locale *sample-locale* can be identified by the presence of a directory named

/usr/lib/locale/*sample-locale*
containing a subdirectory named
/usr/lib/locale/*sample-locale*/LC_MESSAGES
and the following files:
/usr/lib/locale/*sample-locale*/LC_COLLATE
/usr/lib/locale/*sample-locale*/LC_CTYPE
/usr/lib/locale/*sample-locale*/LC_MONETARY
/usr/lib/locale/*sample-locale*/LC_NUMERIC
/usr/lib/locale/*sample-locale*/LC_TIME
The /usr/lib/locale/*sample-locale*/LC_MESSAGES/ subdirectory may in turn contain a series of message files. These are message files for internationalized programs using the AT&T messaging interface (see section "Appendix C: Message catalogs as in AT&T System V, Rel. 4.0" for details).
The default system location for message files for programs based on the X/Open messaging interface is in the directories /usr/lib/nls/msg/ and /opt/lib/nls/msg/ and their subdirectories.
The X/Open messaging interface is flexible, as the NLSPATH environment variable can be used to dynamically control access to message catalogs (see section " X/Open-compliant message catalogs"). That means that message catalogs can be installed at other places in the system.
If NLSPATH is not set or has been assigned an invalid value, messages are taken from the message files in /usr/lib/nls/msg/*language*/ or /opt/lib/nls/msg/*language*/.

Note:

If you create your own databases using the *ic* compiler, the install_locale script that *ic* generates, which has to be run by the system administrator, stores the resultant files in the directories set aside for that purpose (see section "IC - Compiler for international databases").

5.1.3 Storing other NLS-related data

Multiple-locale working often entails multiple-codeset working and the need for conversion between codesets. The iconv command is the right tool for this job. The conversion tables for iconv are stored in the /usr/lib/iconv/ directory as files with names of the form /usr/lib/iconv/*symbol.to_symbol*.t (see below).
The following table shows the minimum set of supported codeset conversions:

Supported codeset conversions				
Code	**Symbol**	**To code**	**To symbol**	**Comment**
ISO 646	646	ISO 8859-1	8859	US ASCII
ISO 646de	646de	ISO 8859-1	8859	German
ISO 646da	646da	ISO 8859-1	8859	Danish

ISO 646en	646en	ISO 8859-1	8859	British ASCII
ISO 646es	646es	ISO 8859-1	8859	Spanish
ISO 646fr	646fr	ISO 8859-1	8859	French
ISO 646it	646it	ISO 8859-1	8859	Italian
ISO 646sv	646sv	ISO 8859-1	8859	Swedish
ISO 8859-1	8859	ISO 646	646	7-bit ASCII
ISO 8859-1	8859	ISO 646de	646de	German
ISO 8859-1	8859	ISO 646da	646da	Danish
ISO 8859-1	8859	ISO 646en	646en	British ASCII
ISO 8859-1	8859	ISO 646es	646es	Spanish
ISO 8859-1	8859	ISO 646fr	646fr	French
ISO 8859-1	8859	ISO 646it	646it	Italian
ISO 8859-1	8859	ISO 646sv	646sv	Swedish

Table 24: Supported codeset conversions

There is a list of all the conversions that the system supports in the file `/usr/lib/iconv/iconv_data`.

5.1.4 SINIX NLS databases

The "SINIX NLS databases" package contains fully defined English and German locales. In the SINIX V5.41 operating system release this package supplies the following codesets:

- the German, British English and American English variants of the ISO 646 7-bit character set
- the ISO 8859-1 8-bit character set
- the ISO 6937 8-bit character set

The two 8-bit character sets can be used in both English and German environments. The ISO 6937 character set is used internally by the COLLAGE user interface. It is a general-purpose character set, but it may not be supported by your hardware.

Note:

For 8-bit character set transmission and for displaying 8-bit characters (such as letters with accents) on screen you need a terminal which is set up and configured as an 8-bit terminal. For details consult your system administrator.

The following table shows the minimum set of selectable locales:

Selectable locales			
Name	**Language**	**Country**	**Character set**
C	English	USA	ASCII
De_DE.646	German	Germany	ISO 646de
En_GB.646	English	Great Britain	ISO 646en
De_DE.88591	German	Germany	ISO 8859-1
En_GB.88591	English	Great Britain	ISO 8859-1
De_DE.6937	German	Germany	ISO 6937
En_GB.6937	English	Great Britain	ISO 6937
En_US.ASCII	English	USA	ASCII

Table 25: Selectable locales

All internationalized programs access the information in these databases. The extent to which individual SINIX commands react to the presence of an NLS database is described in the "Locale" subsection of the associated command description in the manuals "SINIX V5.41 Commands, Volumes 1-3" [1-3].

5.2 Defining the working environment

The SINIX V5.41 operating system supports announcement mechanisms which allow settings for language, national conventions and character set to be defined both on a system-wide basis and for individual users or user groups.
The followings types of environment can be set up:

- single-language working.
  Here the system administrator sets up the same working environment for all the users on a system.
- mixed-language working.
  Here different system users work in different languages.
- multi-language working.
  Here each user can work in a personalized environment, using different languages for different programs. For example, if you are working in a French environment (set with the LANG variable), you can set the LC_COLLATE environment variable to reflect an English locale so that you can sort an English text file in the appropriate collating sequence.
  Note that there is no direct way of sorting a file containing texts in several languages using the collating sequence information appropriate to each language. As the relevant environment variable, LC_COLLATE, is set for the entire working environment, and locale and category assignments are applicable to the current process and any child processes, the system is not capable of distinguishing between the different parts of the text.
  However, it is conceivable that a user faced with such requirements could design a custom collating sequence containing the essential elements of the various languages (see chapter "Localizing internationalized software"). Even then it would have to be possible to represent all parts of the text in a single character set.

You set your working environment by assigning the name of the required locale to a set of environment variables reserved for that purpose. Whenever an internationalized program is invoked, these environment variables are read and the information for the locale assigned to them is bound to the program's runtime environment.
The environment variables which set the locale are:

```
LANG
LC_ALL
LC_MESSAGES
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
NLSPATH
TZ
```

The order of precedence among these variables is indicated in the table "Precedence among environment variables".
There are two forms of assignment to these variables (taking LANG as an example):

LANG=*locale-name*

LANG=*language_territory.codeset*

The LC_CTYPE, LC_COLLATE, LC_TIME, LC_MONETARY, LC_NUMERIC and LC_MESSAGES variables also support an additional @*modifier* element specifying extra alternatives, for example a particular type of collating sequence.

LC_CTYPE=*language_territory.codeset@modifier*
The variable parts of this assignment (*language_territory.codeset@modifier*) are file names in /usr/lib/locale and its subdirectories. The length of this string must not be greater than the {NL_LANGMAX} system constant.

Example
In a German environment you can use @*modifier* in your assignment to LC_COLLATE to choose between the standard collating sequence (dictionary order) and the collating sequence preferred in the German telephone directory. The collation rules here differ in respect of uppercase and lowercase, umlauts and special characters. To select the phone book collating sequence you use the assignment:
LC_COLLATE=De_DE.88591@TE
In this collating sequence lowercase and uppercase are treated as identical, ä is treated as if it were ae, ö as oe, ü as ue, ß as ss, Ä as Ae, Ö as Oe, and Ü as Ue. Where words differ only with regard to case, the following sequence applies: a, A, b, B, ... z, Z.
Where words differ only with regard to representation of umlauts or use of ss/ß, the following sequence applies:
ae, ä, Ae, Ä, AE
oe, ö, Oe, Ö, OE
ue, ü, Ue, Ü, UE
ss, ß
If @*TE* is not specified, the following collating sequence applies to letters:
- with En_GB.646, En_GB.6937 and En_GB.88591: a, b, ... z, ... A, B, ... Z
- with De_DE.646, De_DE.6937 and De_DE.88591: lowercase before uppercase (a, ... z, A, ...Z), with ä treated as a, ö as o, ü as u, ß as ss, Ä as A, Ö as O and Ü as U. Where words differ only with regard to representation of umlauts or use of ss/ß, the applicable sequence is: a, ä, b, c, ... , ss, ß, ... z, A, Ä, B, ... Z.

If any of the variables LC_CTYPE, LC_COLLATE, LC_TIME, LC_MONETARY, LC_NUMERIC or LC_MESSAGES is undefined or is assigned the null string, its value defaults to the value of LANG.
If any of the variables LC_CTYPE, LC_COLLATE, LC_TIME, LC_MONETARY, LC_NUMERIC or LC_MESSAGES has an invalid value, or if the LANG variable is undefined or null, the system acts as if it were not internationalized, collating in the order of the characters in the ASCII character set. This is the default setting for the SINIX V5.41 operating system.

5.2.1 LANG environment variable

Every value of the LANG environment variable is associated with a complete locale, i.e. defined collating sequence, character classifications and conversions, date and currency formats, numeric formatting information and message catalogs. LANG allows you to set values for *language*, *territory* and *codeset* for locales with names complying with the X/Open conventions.
On shell level you assign the LANG variable the name of the locale you require (the required language, territory and codeset) in one of the following forms:
LANG=*locale-name*
LANG=*language_territory.codeset*
The accepted values are the names of subdirectories of /usr/lib/locale (see section "Mandatory strings by category "). The length of this string must not be greater than the

{NL_LANGMAX} system constant.
Like any environment variable, LANG can be preset for the entire system by the system administrator. However, every user can also define a personal setting for it at any time.
LANG provides a general announcement mechanism that allows users to identify overall requirements for program localization. This is sufficient when a single localization covers all a user's requirements for text collation, character classification and message presentation.
Thus on the SINIX V5.41 operating system specifying

```
LANG=De
```

is enough, thanks to a symbolic link to `/usr/lib/locale/De_DE.646`, to select German as the language, German conventions and the ISO 646de character set.
Similarly, specifying

```
LANG=De_DE.88591
```

selects German language and conventions and the ISO 8859-1 character set.
If this assignment is made on a system-wide basis by the system administrator, it applies to all users and sets up a single-language working environment.
Regardless of the setting defined by the system administrator, as a user you can set up your own personal working environment on the system. This entails including the required assignment to LANG in your `/$HOME/.profile` file. As your `.profile` is interpreted after the `/etc/default/language` file in which the system administrator's setting of the LANG variable is stored, the environment you require will be initialized whenever you log in or invoke a shell.
If you want to change the working environment within a shell, e.g. between one command and the next, you can also set the LANG environment variable directly from the command line.
Setting LANG directly on the command line or by way of an entry in a personal `.profile` allows a multiuser system to support mixed-environment working.

5.2.2 LC_xxx environment variables

Like the LANG variable, the LC_ALL environment variable provides a general announcement mechanism for the entire locale, and it uses the same syntax. Unlike LANG, LC_ALL has the top precedence, which means that it overrides all other international environment variables. Setting LC_ALL is sufficient if one preset locale satisfies all user requirements for program localization.
LANG has the lowest precedence; so if it is set, it still possible to set other international environment variables to customize the working environment of individual users to meet their specific requirements.

Precedence	**Environment variable**
high	LC_ALL
medium	LC_CTYPE
	LC_COLLATE
	LC_TIME
	LC_MONETARY

	LC_NUMERIC
	LC_MESSAG ES
low	LANG

Table 26: Precedence among environment variables

A typical case not covered by either LANG or LC_ALL is where you want to communicate with the system in one language but sort text files, for example, in another. In cases of this type, you define the environment variables in the table below, which allow you to modify individual aspects (categories) of your locale.

Environment variable	**affects**
LC_CTYPE	Character classes and upshifting/downshifting
LC_COLLATE	Collating sequence
LC_TIME	Date and time strings
LC_MONETARY	Currency symbols and monetary formatting information
LC_NUMERIC	Form of radix character, exponentiation sign and digit grouping symbol
LC_MESSAGES	Message texts and replies to yes/no questions

Table 27: Locale variables

This if you want to communicate with the system in British English but you also need to sort German text files, you should define the following settings for the LANG and LC_COLLATE environment variables:

```
LANG=En_GB.88591
LC_COLLATE=De_DE.88591
```

By setting other variables you can set up a multi-language working enironment (see section " Setting up a multi-language working environment").

Note:

Note that the LC_ALL environment variable must be left unset in such cases, as it overrides all other international environment variables. If LC_COLLATE and LC_ALL are both set, LC_COLLATE is ignored.

5.2.3 NLSPATH environment variable

This variable has an effect similar to that of PATH (see `sh` in the manual "SINIX V5.41 Commands, Volume 2" [2]). In it you specify the path names of the message catalogs from which internationalized programs are to retrieve their message texts at runtime. The value of this variable takes the form of a colon-separated list of path names. The paths are then scanned from left to right. The path names may include the following metacharacters:

Metacharacte	**expands to**

r	
%L	$LANG
%l	*language* component of $LANG
%t	*territory* component of $LANG
%c	*codeset* component of $LANG
%N	parameter of the `catopen()` function, generally the name of the program/command whose message catalog is being searched for

Table 28: Metacharacters in path name

The presence of :: or :.: in the value of NLSPATH causes a file named %N to be looked for in the current directory. If the value of NLSPATH begins with a colon :, a file named %N will be searched for in the current directory first.

Example

`NLSPATH=:/usr/lib/nls/msg/%L/%N.cat:/`*`my_dir`*`/%N.cat`
If LANG has the value *De* (for German) and you call the `date` command, the message catalog `date.cat` will be looked for first in the current directory, then in the directory `/usr/lib/nls/msg/De`, and finally in the directory `/`*`my_dir`*.
If the message catalog cannot be found under the path name assigned to NLSPATH, or if NLSPATH is not set, the default is to look for it in `/usr/lib/nls/msg/%l/%N.cat` and then in `/opt/lib/nls/msg/%l/%N.cat`.

5.2.4 TZ environment variable

The TZ environment variable affects timezone information. The `date` command and most of the library functions which process time data use TZ to determine the timezone and to convert UTC (Universal Time Coordinated, same as Greenwich Mean Time) to local time and vice versa.
The following files (names of timezones or countries) reside in `/usr/lib/locale/TZ/`:

CET	CST6 CDT	EET	EST	EST5 EDT	GB-Eire	GMT
GMT+1	GMT+2	GMT+2	GMT+3	GMT+4	GMT+5	GMT+6
GMT+7	GMT+8	GMT+9	GMT+10	GMT+11	GMT+12	GMT+13
GMT-1	GMT-2	GMT-3	GMT-4	GMT-5	GMT-6	GMT-7
GMT-8	GMT-9	GMT-10	GMT-10	GMT-11	GMT-12	Greenwich
HST	Hongkong	Iceland	Israel	Japan	MET	MST
MST7 MDT	NZ	Navajo	PRC	PST8 PDT	Poland	ROC

ROK	Singapore	Turkey	UCT	UTC	Universal	W-SU
WET						

Table 29: Timezones for the TZ variable

The same directory also has the following subdirectories (names of countries extending over a number of timezones), supporting use of the *:zonename* syntax for the TZ variable (see the manuals "SINIX V5.41 Commands, Volumes 1-3" [1-3]).

Directory	**Contents**		
Australia	Australia/Broken-Hill Australia/LHI Australia/NSW Australia/North	Australia/Queensland Australia/South Australia/Sturt Australia/Tasmania	Australia/Victoria Australia/West Australia/Yancowinna
Brazil	Brazil/Acre Brazil/West	Brazil/DeNoronha	Brazil/East
Canada	Canada/Atlantic Canada/Central Canada/East-Saskatchewan	Canada/Eastern Canada/Mountain Canada/Newfoundland	Canada/Pacific Canada/Yukon
Chile	Chile/Continental	Chile/EasterIsland	
Mexico	Mexico/BajaNorte	Mexico/BajaSur	Mexico/General
US	US/Arizona US/Central US/East-Indiana	US/Eastern US/Hawaii US/Mountain	US/Pacific US/Pacific-New US/Yukon

Table 30: Timezone names for the TZ variable

For information on the contents of these files and how to generate similar files refer to the description of the `zdump` and `zic` commands in "SINIX V5.41 System Administrator's Reference Manual" [4].

5.2.5 Setting up a single-language working environment

On a system with no environment variables set, programs operate in accordance with the conventions of a default locale. This default setting may vary from system to system; so system administrators and users alike should check whether the setting is appropriate to their requirements with regard to their working environment. The default locale on the SINIX V5.41 operating system is "C".
The system administrator is able to change this default setting globally for all system users by setting the relevant international environment variables in the startup files for the shell they use.

Example

If `sh` is being used as the command interpreter, the `/etc/default/language` file might contain the following assignment:

```
LANG=De_DE.88591
```

This would set the language and the cultural conventions to German and the character set to 8859-1. As already explained, LANG supports a mechanism for defining all locale requirements.

Note:

As a UNIX/SINIX system administrator you can assign users different command interpreters, each interpreting different startup files. Therefore you must make sure that the environment are set in all the relevant startup files.

5.2.6 Setting up a mixed-language working environment

The active working environment for every process on a UNIX/SINIX system is governed by environment variables. That means that different settings of the working environment can be defined independently for each process. Thus each system user can set up a working environment based on a different language, which will impact on all child processes the user spawns.
If, for example, `sh` is being used as the command interpreter, the usual approach is to set the appropriate environment variables in the `/$HOME/.profile` startup file. The user's `.profile` is executed after the `/etc/default/language` file in which the system administrator defines global environment variable settings. As a result the working environment a user requires is set up every time that user logs in or invokes a shell.
A shell user wanting to change the working environment between one command and the next, for example, can redefine the environment variables directly from the command line.

5.2.7 Setting up a multi-language working environment

The term "multi-language working environment" refers to user-defined setting of specific categories within a working environment on the system; in other words, a user can specify that certain parts of an application are directly assigned to categories from different locales.
This entails setting the category-specific environment variables LC_CTYPE, LC_COLLATE, LC_TIME, LC_MONETARY, LC_NUMERIC and LC_MESSAGES. If these variables are not set, the information for the associated categories is taken from the locale defined by LANG. For example, if you need to have system messages output in a different language, you can do so by assigning a different locale to the LC_MESSAGES variable.

Example

```
$ ls -CF /usr/lib/locale
C/   En_US.ASCII/  En_GB.646/  Fr_FR.88591/  De_DE.88591/
$ LANG=En_US.ASCII# sets the default locale
# to English
$ date
Tue Jul 14 22:06:15 MET 1992        Date is displayed in English
$ LC_TIME=Fr_FR.88591# sets the date component
```

```
# of the locale to French
$ date
Mar 14 Juil 1992 MET 22:07:06      Date is displayed in French
$ foo
foo: not found                     Messages remain in English
$ LC_MESSAGES=De_DE.88591# sets the text component of
# the locale to German
$ foo
foo: nicht gefunden                Messages are now displayed in German
$ date
Mar 14 Juil 1992 MET 22:08:57      Date output remains in French
$ LC_ALL=De_DE.88591# sets the entire locale to
# German
$ date
Di.14.Jul.1992, 22:09:38 MEZ       Date now appears in German
```

6 Appendix

6.1 Appendix A: Interface classification

The following tables are an overview of the origins of Siemens Nixdorf Informationssysteme AG internationalization interfaces (library functions, commands and header files). The sources referred to are the X/Open XPG3 and XPG4 standard and the de facto UNIX SVR 4.0/4.2 standard (SVR4). The interfaces summarized here are guaranteed on a long-term basis by the SINIX API (API, refer to "SINIX API Application Programming Interfaces, Interfaces Platform"[7]).

Single-byte library functions

Function	Source	Function	Source	Function	Source
atof	XPG3	catclose	XPG3	catgets	XPG3
catopen	XPG3	fprintf	XPG3	fscanf	XPG3
gcvt	XPG3	getdate	SVR4	gettxt	SVR4
isalnum	XPG3	isalpha	XPG3	iscntrl	XPG3
isgraph	XPG3	islower	XPG3	isprint	XPG3
ispunct	XPG3	isspace	XPG3	isupper	XPG3
localeconv	XPG4	nl_langinfo	XPG3	printf	XPG3
scanf	XPG3	setlocale	XPG3	sprintf	XPG3
sscanf	XPG3	strcoll	XPG3	strftime	XPG3
strtod	XPG3	strxfrm	XPG3	tolower	XPG3
toupper	XPG3	vfprintf	XPG3	vprintf	XPG3
vsprintf	XPG3				

Table 31: Interface classification for single-byte library functions

XPG3 in the above table indicates that the functions comply with both the XPG3 standard and the XPG4 standard.

Multibyte library functions

Function	Source	Function	Source	Function	Source
fgetwc	XPG4	fgetws	XPG4	fputwc	XPG4
fputws	XPG4	getwc	XPG4	getwchar	XPG4

getws	XPG4	iswalnum	XPG4	iswalpha	XPG4
iswcntrl	XPG4	iswdigit	XPG4	iswgraph	XPG4
iswlower	XPG4	iswprint	XPG4	iswpunct	XPG4
iswspace	XPG4	iswupper	XPG4	iswxdigit	XPG4
mblen	XPG4	mbstowcs	XPG4	mbtowc	XPG4
putwc	XPG4	putwchar	XPG4	putws	XPG4
towlower	XPG4	towupper	XPG4	wcscat	XPG4
wcschr	XPG4	wcscmp	XPG4	wcscoll	XPG4
wcscpy	XPG4	wcscspn	XPG4	wcsftime	XPG4
wcslen	XPG4	wcsncat	XPG4	wcsncmp	XPG4
wcsncpy	XPG4	wcspbrk	XPG4	wcsrchr	XPG4
wcsspn	XPG4	wcstod	XPG4	wcstok	XPG4
wcstol	XPG4	wcstombs	XPG4	wcstoul	XPG4
wcswcs	XPG4	wcswidth	XPG4	wcsxfrm	XPG4
wctomb	XPG4	wcwidth	XPG4		

Table 32: Interface classification for multibyte library functions

Commands

Command	Source	Command	Source	Command	Source
chrtbl	SVR4	colltbl	SVR4	dumpmsg	API
exstr	SVR4	extract	API	fmtmsg	SVR4
gencat	XPG4	gettxt	SVR4	ic	API
iconv	XPG4	iecho	API	iput	API
mkmsgs	SVR4	montbl	SVR4	srchtxt	SVR4
zdump	SVR4	zic	SVR4		

Table 33: Interface classification for commands

Header files

Header file	Source
ctype.h	XPG4
langinfo.h	XPG4
limits.h	XPG4
locale.h	XPG4
nl_types.h	XPG4

Table 34: Interface classification for header files

6.2 Appendix B: Code tables

Latin Alphabet No. 1 (ISO 8859-1)

b_4 b_3 b_2 b_1	b_8 b_7 b_6 b_5	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0 0 0 0	00			SP	0	@	P	`	p			NBSP	°	À	Ð	à	ð
0 0 0 1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
0 0 1 0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
0 0 1 1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
0 1 0 0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
0 1 0 1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
0 1 1 0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
0 1 1 1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷
1 0 0 0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø
1 0 0 1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù
1 0 1 0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
1 0 1 1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û
1 1 0 0	12			,	<	L	\	l	|			¬	1/4	Ì	Ü	ì	ü
1 1 0 1	13			–	=	M	]	m	}			SHY	1/2	Í	Ý	í	y
1 1 1 0	14			.	>	N	^	n	~			®	3/4	Î	Þ	î	þ
1 1 1 1	15			/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ

Figure 3: Latin Alphabet No. 1 (ISO 8859-1)

International 7-bit code (ISO 646-IRV)

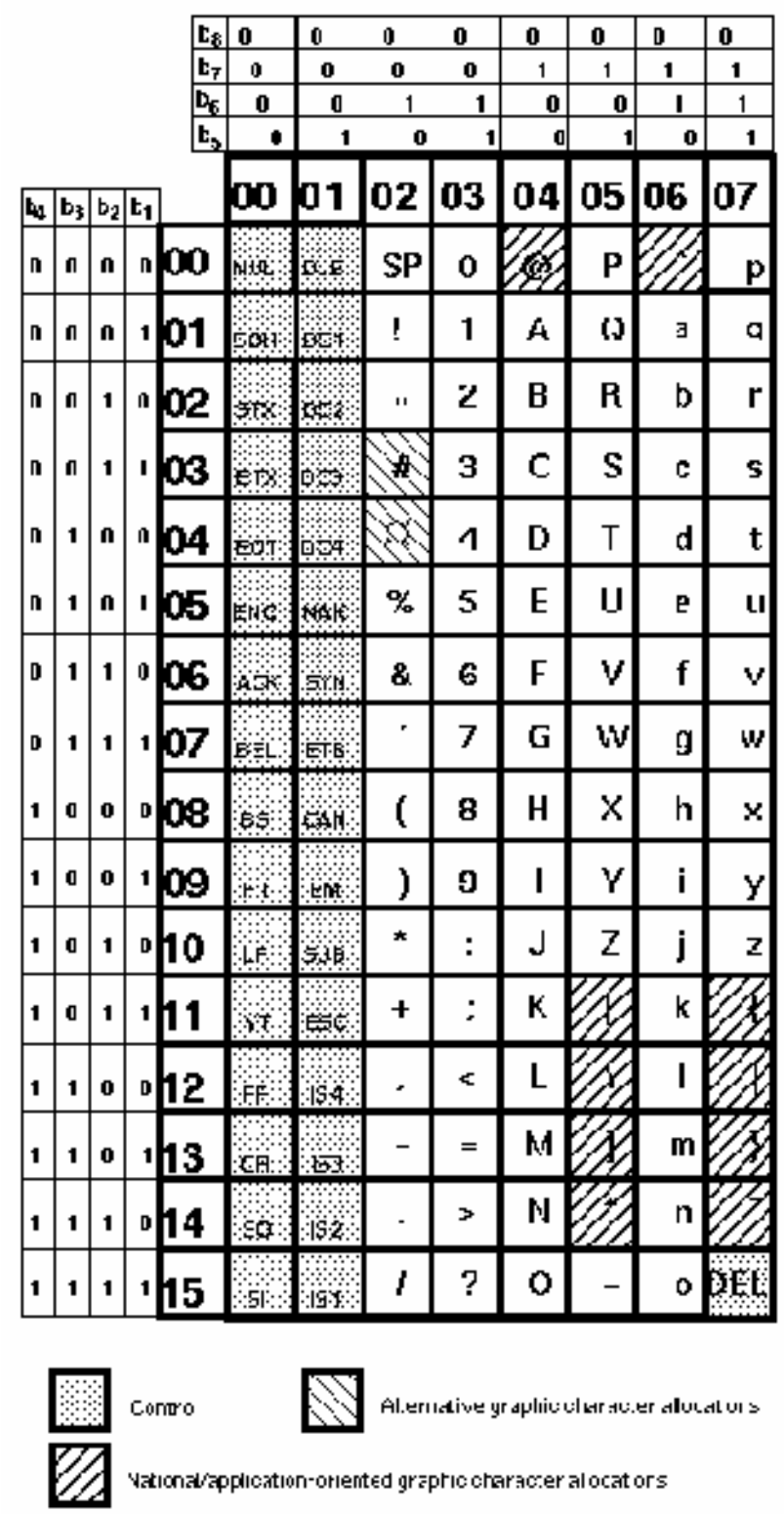

Figure 4: International 7-bit c ode (ISO 646-IRV)

6.3 Appendix C: Message catalogs as in AT&T System V, Rel. 4.0

SINIX V5.41 systems by default support both the X/Open messaging system and the AT&T messaging system. Essentially it is possible to generate message catalogs using both systems; but it is best to use only one of them on any one system.

6.4 Commands

The following commands are used by the AT&T messaging system.

extr	fmtmsg	gettxt
mkmsgs	srchtxt	

Table 35: Commands of the AT&T messaging system

exstr
used to generate internationalized programs. This command allows you to search for strings in C programs and replace them with function calls.
gettxt
looks in a message file created by `mkmsgs` for the string allocated to a given message identifier.
fmtmsg
writes formatted messages to standard error or the system console.
mkmsgs
creates a message file which can be accessed by the text retrieval tools `gettxt, srchtxt` and `exstr` and the C function `gettxt()`. This file has a different format to that interpreted by the X/Open commands `gencat` and `dumpmsg`.
srchtxt
displays the contents of message catalogs, or searches for specified text strings in message catalogs.

C library function

`gettxt()` acts in the same way as the `gettxt` command.

Actions to be taken by the programmer

As a programmer you should proceed as follows in order to generate an internationalized application:
- keep the message text separate from the logic of the program by using the `gettxt()` function, which retrieves the messages at program runtime.
- create a message text source file in the defined format (for format see below).
- generate a message catalog from the message text source file using the `mkmsgs` command.

It is then possible to create other internationalized environments by translating the message text source file into other languages and compiling the files using `gencat` or `mkmsgs`.

Format

Message text source files to be compiled by `mkmsgs` have to be in the following format:
- the messages are not divided into sets and are not identified by message identifiers,
- each message string has to be on a separate line,
- unlike in the X/Open format, the messages have to be in ascending and contiguous order, as the line numbers are used to identify the messages,
- non-printing characters must be represented in the form of escape sequences.

Example

```
This is message 1
This is message 2
```

```
This is message 3
This is message 4
```

Accessing

Message catalogs are accessed by calling the library function `gettxt()`, which retrieves the message with the specified *messageid* from the given file.
The `gettxt()` function searches for the given file in the directory `/usr/lib/locale/<`*locale-name*`>/LC_MESSAGES`. The language environment (*locale-name*) accessed is governed by the value of the LC_MESSAGES environment variable. If this variable is undefined, the default is the value of the LANG environment variable. If LANG is not set either, the default directory is `/usr/lib/locale/C/LC_MESSAGES`, which contains default message texts.
messageid denotes the number of the line containing the required message string.
The `gettxt()` function is equivalent to the `catgets()` function in the X/Open messaging system; but the AT&T messaging system has no functions equivalent to `catopen()` and `catclose()`.

Example

```
printf( gettxt("UX:10","error-text"));
```

This statement attempts to retrieve the message in line 10 of file UX in the directory `/usr/lib/locale/<`*locale-name*`>/LC_MESSAGES`. If it fails to find the message, it outputs the string *error-text*.

Note:

As the X/Open messaging system's `extract` command can be parameterized by means of a pattern file, it can also be used for the AT&T messaging system. This does, however, entail creating an AT&T-format pattern file (see section "Generating message catalogs").

For a full description of the `extract` command refer to the manual "SINIX V5.41 Commands, Volume 1" [1].

7 References

Siemens Nixdorf Informationssysteme AG publications

[1] **Commands** (SINIX V5.41)
Part 1, A - K
Reference Manual
Target group
SINIX shell users
Contents
Alphabetically arranged description of the SINIX command set

[2] **Commands** (SINIX V5.41)
Part 2, L - Z
Reference Manual
Target group
SINIX shell users
Contents
Alphabetically arranged description of the SINIX command set

[3] **Commands** (SINIX V5.41)
Part 3, Reference Section
Reference Manual
Target group
SINIX shell users
Contents
Tables and reference section for commands described in Parts 1 and 2
- Table of contents
- Command overview
- Regular expressions
- Bourne shell metacharacters
- Data media special files
- SINIX V5.23 and V5.40 SPOOL system files
- ISO 646 character set
- References and index

[4] MX500 (SINIX V5.40)
MX300 (SINIX V5.41)
System Administrator's Reference Manual
Target group
System administrators
Contents
Describes commands and application programs for system maintenance, file formats and special system administration files and provides notes on diagnostics

[5] **CES** (SINIX)
Guide to Tools for Programming in C
User Guide

Target group
This manual is intended for use by C programmers working in a SINIX operating system environment.
Contents
The manual describes the C compilation system (preprocessor, link editor, header files, libraries), and utilities for developing, maintaining and generating C programs.

[6] C-DS C V1.0
Programmers Reference Manual
Target group
C-programmers working with C-DS C Version 1.0.
Contents
Description of commands, system calls, functions, header files and C-specific file formats for the developement of C-programs.

[7] **SINIX API**
Interfaces Platform
Reference Manual
Target group
Data processing organizers and planners; software developers
Contents
The SINIX API interfaces platform defines on a long-term binding basis those Siemens Nixdorf open systems software interfaces which are important for application programming.

[8] Siemens Nixdorf
IHB Internationalization Handbook
Guidelines for the Manufacture of International Products
Reference Manual
1992

[9] Siemens Nixdorf
IHB Checkliste
Checklist for the Internationalization Handbook
Catalog of Questions
1992

[10] Siemens Nixdorf
IHB Tables / IHB Tabellen
Supplement for Internationalization Handbook /
Ergänzungen zum Internationallisierungshandbuch
Ready Reference / Tabellenbuch
1992

Other publications

[11] X/Open Guide
XPG3-XPG4 Base Migration Guide
ISBN: 1-872630-49-9
X/Open Company Ltd.
Reading, July 1992

[12] X/Open Guide
System Interfaces and Headers, Issue 4
ISBN: 1-872630-47-2
X/Open Company Ltd.
Reading, July 1992
[13] X/Open Guide
System Interface Definitions, Issue 4
ISBN: 1-872630-46-4
X/Open Company Ltd.
Reading, July 1992
[14] X/Open Guide
Internationalisation Guide
ISBN: 1-872630-20-0
X/Open Company Ltd.
Reading, February 1992

Ordering manuals

The manuals listed above and the corresponding order numbers can be found in the Siemens Nixdorf *List of Publications*. New publications are described in the *Druckschriften-Neuerscheinungen* (*New Publications*).
You can arrange to have both of these sent to you regularly by having your name placed on the appropriate mailing list. Please apply to your local office, where you can also order the manuals.

Glossary

8 **8-bit transparency**

8-bit transparency refers to the ability of a software product to process characters from 8-bit character sets without modifying or utilizing any part of the character in a way that is inconsistent with the rules of the current coded character set. All eight bits of a byte are treated as part of the character code.

A **ASCII**

American Standard Code for Information Interchange. A 7-bit coded character set designed for the English language, consisting of 128 characters with no umlauts or accented letters.

B **byte**

A unit of information consisting of 8 bits used to represent graphical information such as printing characters and control data affecting the way text is recorded, processed and interpreted.

C **category**

Classification of country-specific information within a locale in information groups. The LC_COLLATE category, for example, covers all sorting (collating sequence) information. Categories provide flexible selective access to country-specific information

character conversion

The process of converting data coded for one character set to data coded for another character set.

character set

A set of alphanumeric or other characters designed to represent information. These characters are used to construct the words and other elementary units of a native language or computer language.

code table

A list in table form of the bit combinations allocated to the characters in a character set.

country-specific conventions

Information defining the formats a locale uses to represent the date/time and numeric and monetary quantities. These formats may vary from locale to locale.

D **database**

In relation to internationalization the term database refers to the sum total of country-specific information held in national libraries (national databases). Such information is kept separate from the neutral core of an internationalized program.

diphthong

A linguistics term for a compound sound formed by two vowels. In the field of internationalization, diphthongs affect character collating sequences. Collating sequences

applicable to internationalized programs must assign diphthongs 1-to-2 mappings in the database's collation table.

I

ideogram
A graphic symbol representing an idea rather than a word. In an ideographic script the number of symbols is roughly equal to the number of ideas expressible in the associated language. Ideographic scripts are used by languages such as Chinese, Japanese and Korean.

internationalization
The term internationalization ideally refers to the process of designing software without making assumptions about the language, code tables, or national and local conventions of the environment in which the program will be used. On system level that means that internationalized programs link in the country-specific information they need from national databases at runtime using specially designed interfaces.

ISO
International Organization for Standardization. An international association of autonomous national standard-setting organizations working together to establish global standards.

L

locale
The sum total of language and country-specific information for a particular language variant. This determines the working environment in which a product is used. It includes information on language, character set, collating sequence, character classification and conversion, on the language of the message catalogs and on cultural conventions.

localization
The process of establishing information within a computer system specific to the language-specific and country-specific needs of a particular user or user group. This process does not alter the functionality of the product.

M

message catalog
A file holding program-dependent information such as program messages, command prompts and responses to prompts for a particular native language.

messaging system
A system designed to generate, store, update and access program text used for interaction between the software and the user. A messaging system allows interaction to take place in a variety of different languages.

multibyte character
A single character represented in a character set by a sequence of two or more bytes.

N

national database
A national database contains the sum of all the country-specific data designed for use in a particular location (see locale).

Native Language System
A set of language-neutral and script-neutral function interfaces supporting development of international software products in which the language and country-specific data is kept separate from the neutral product core.

neutral product core
The uniquely identifiable product component which is independent of a particular location's conventions for language, script, and cultural and regional data and does not need to be modified when country-specific versions of a product are generated.

NLS
see Native Language System.

NLSPATH
An environment variable used in UNIX to indicate the search path for message catalogs.

P Portable Character Set

To ensure data portability to all POSIX-compliant systems, only characters from the Portable Character Set should be used. This set comprises the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 ! # % ^ & * ( ) _ + - = { } [ ]
: " ~ ; ' ' < > ? , . | \ / @ $
```

Portable Filename Character Set
To ensure file portability to all POSIX-compliant systems, the characters used in file names should be limited to those in the Portable Filename Character Set, which comprises the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
```

POSIX
Portable Operating System Interface for Computer Environments. An IEEE (Institute for Electrical and Electronics Engineers) set of standards relating to software product portability. The name POSIX is often used to refer to the 1003.1 Standard, which defines an interface for operating systems.

X X/Open

An international association of hardware and software vendors with the aim of creating a free and open software market for UNIX applications.

Index