ULTRIX

digital

Guide to Developing International Software

Order Number: AA–LY26B–TE

ULTRIX

Guide to Developing International Software

Order Number: AA-LY26B-TE
June 1990

Product Version: ULTRIX Version 4.0 or higher

digital equipment corporation
maynard, massachusetts

Contents

Figures

Tables

About This Manual

The ULTRIX Internationalization package provides tools and functions to allow you to write software that can be used in a number of nations. The program interface appears to users in each nation as if designed for that nation's users. For example, messages appear in the native language of the user and the full character set for the user's language is available.

Audience

This guide is intended for experienced ULTRIX application programmers writing software intended for multinational or non-English language use. Translators who translate the messages displayed by international software might also find this guide useful. Application programmers should read this entire guide in conjunction with the internationalization package reference pages. Translators should find Chapter 1, Chapter 2, Appendix B, and the `trans`(1int) translation editor reference page the most useful.

Organization

This guide consists of four chapters, three appendixes, and a glossary.

Chapter 1	Internationalization Overview Introduces the basic concepts and components of the ULTRIX internationalization package.
Chapter 2	The Message Catalog System Describes message catalogs, international library routines, and the associated tools you use to generate and fill them.
Chapter 3	Program Localization Explains how the language requirements of international software are announced to the system.
Chapter 4	Language Support Databases Describes the language support databases that allow programs to operate in various native languages, and the source language used to create input files for the `ic` compiler.
Appendix A	Database Source Language Syntax Description Gives an Extended Backus-Naur Form (EBNF) notation of the syntax recognized by the `ic` compiler.
Appendix B	An Example of a Source Language File Gives an example source file for a language support database.

Appendix C	A List of Associated Reference Pages Lists the ULTRIX reference pages associated with the internationalization package.

Conventions

The following conventions are used in this manual:

%	The default user prompt is your system name followed by a right angle bracket. In this manual, a percent sign (%) is used to represent this prompt.
user input	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	This typeface is used in interactive examples to indicate system output and also in code examples and other screen displays. In text, this typeface is used to indicate the exact name of a command, option, partition, pathname, directory, or file.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
rlogin	In syntax descriptions and function definitions, this typeface is used to indicate terms that you must type exactly as shown.
filename	In examples, syntax descriptions, and function definitions, italics are used to indicate variable values; and in text, to give references to other documents.
[ ]	In syntax descriptions and function definitions, brackets indicate items that are optional.
. . .	In syntax descriptions and function definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
`cat`(1)	Cross-references to the *ULTRIX Reference Pages* include the appropriate section number in parentheses. For example, a reference to `cat`(1) indicates that you can find the material on the `cat` command in Section 1 of the reference pages.
RETURN	This symbol is used in examples to indicate that you must press the named key on the keyboard.

Internationalization Overview 1

The ULTRIX Internationalization package provides tools and functions to enable the internationalization of programs and the environment in which they operate.

Internationalization is the process of designing or adapting programs to meet international requirements, such as those of multiple local languages and the specific character sets associated with them.

1.1 The Purpose of Internationalization

An internationalized program:

- Allows users to interact with the program in their own languages
- Reflects the culture of the users' regions

Conventions for representing cultural data can vary from one country to another and from region to region within a single country. For example:

- Number representation

  England and France both represent numbers using radix characters and commas, but these symbols are interchanged (2,345.77 in England and 2.345,77 in France).

- Currency symbols

  In Italy five thousand Lire is represented by L. 5.000 and in Greece five thousand drachmae is represented by 5,000 Dr.

- Date order

  October 7, 1986 would be represented as 10/7/1986 in the U.S., 7.10.1986 in Germany, and 1986/10/7 in Japan.

- Codeset

  In Switzerland, a program might have to run using Italian, German, and French codesets (modified for local use).

You can meet these internationalization requirements by writing programs that make no assumptions about language, local customs, or coded character sets. Such a program is said to be *internationalized*. Data specific to any particular language, including cultural data, and the codeset, are held separate from the program logic. The process of establishing such data is referred to as *localization*. Run-time facilities bind a program to the appropriate language for its message text.

1.2 The ULTRIX Internationalization Solution

The ULTRIX Internationalization solution consists of:

- Message catalogs and associated tools
- A set of library routines
- Internationalized interface definitions of standard C library routines
- An announcement mechanism
- Language support databases
- An international compiler for the database

The message catalogs are simple databases that enable the program messages to be held externally to the program. The tools are used to assist in the extraction and translation from one language to another of the message text, and to generate message catalogs.

The set of library routines enables programs to determine cultural and language-specific data dynamically (for example the format of date and time strings, day and month names, currency symbols, radix character symbols).

The internationalized interface definitions provide language-dependent character type classification, conversion from uppercase to lowercase and lowercase to uppercase, date and time messages, floating point to string conversions, and text collation.

The announcement mechanism identifies the national language, local custom (territory), and codeset requirements (referred to as ''language'' in the remainder of the guide) appropriate to each user for applications at runtime. Language support databases contain the tables that hold the language-specific data, with one database for each supported language.

The international compiler `(ic)` supplied with the internationalization package compiles the source languages information into the language support databases.

1.2.1 International Keyboard Support

Programmers writing applications that support several languages must take into account that languages are represented within the system by the characters of one or more coded character sets. Because of the requirements of different languages, the coded character sets may vary in both size and representation.

In the international environment, you need to use characters that your coded character set does not use. You can create characters that do not exist as standard keys on your keyboard by using compose sequences. A compose sequence is a series of keystrokes that creates a character. You can create any character from the character set your terminal or DECterm session (if you are using ULTRIX Worksystem Software) is currently using.

Depending on your keyboard, you compose characters in either of the following ways:

- You use three-stroke sequences for a VT320 keyboard
- You use two-stroke sequences on all keyboards except the North American/United Kingdom, the Dutch, and the Norwegian/Danish keyboards, which all use three-stroke sequences.

For more information on composing characters, see the hardware manual that came with your terminal or the *DECwindows Desktop Applications Guide* if you are using ULTRIX Worksystem Software.

The Message Catalog System 2

The message catalog system allows users to interact with the program in their language. The program message text is stored in a message catalog separate from the main body of the program. Message catalogs can be translated into several languages to meet the language requirements of each user.

This chapter describes:

- How to create a message catalog
- How to use the string extraction tools to extract text strings from a program source file, and to replace the extracted strings with library routines
- The format of the message text source file produced
- How to translate the message text strings in the message text source file
- How to use `gencat` to produce a message catalog containing the translated messages

Accessing a message catalog is covered in Section 2.5.1. The access mechanism retrieves a message catalog at run-time and binds it to a particular program. Each internationalized program contains a number of library routines. The library routines retrieve the message text from the message catalog. Library routines are described in Section 2.5.2.

The routine used for accessing the opened catalogs is `catgets`. This routine retrieves messages from a message catalog opened by a call to `catopen`. The routine `catclose` closes an open message catalog.

In the message catalog system, message source files are suffixed by a `.msf` and message catalog files are suffixed by a `.cat`.

2.1 Creating a Message Catalog

To create a message catalog:

1. Write the program, including the program messages.
2. Use the string extraction tools to extract the message text and put it in a message text source file.
3. Translate the message text source file into the required national languages using `trans`.
4. Pass the message text source files through `gencat` to create the message catalogs.

All these steps are described in this chapter. Any text editor can be used to create the program source file.

You can combine Steps 1 and 2 if the source program includes the calls to the message catalog retrieval functions. In this case, the `catgets` or `catgetmsg` routines should be included in the source file as appropriate. The message text string can then be extracted using a stream editor and stored in the message text source file.

Message catalogs can be divided into one or more sets of program messages, each set containing one or more messages. The library routines allow programs to access messages within message sets.

The internationalization tools used to create a message catalog are:

- `extract` for interactive message string extraction
- `strextract` for batch message string extraction
- `strmerge` for batch message source file merging (used in conjunction with `strextract`)
- `trans` translation tool
- `gencat` message catalog generator

Each of these tools is described on the relevant reference page, for example `extract`(1int) utility. The information in this section about these tools supplements that contained on the reference pages.

2.2 String Extraction

You can use the string extraction tools to partially automate the process of internationalizing a C program. For example, you could use them to change the following segment from a C program:

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

into

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

and the corresponding message text source into

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

There are two ways to extract text strings from a particular program source file and to replace the extracted strings with library routines:

- Use the interactive extraction tool `extract` on its own
- Use the batch extraction tool `strextract` followed by the batch merging tool `strmerge`

In both cases the extracted message text is stored in a message source file suffixed `.msf`. The message text can then be translated using the `trans` translation tool.

The translated messages in the source file are submitted to `gencat` to generate a message catalog. At run-time, the library routines in the internationalized program retrieve the translated text from the message catalog.

The interactive and batch methods of string extraction use the following files:

- A pattern file

- An optional ignore file
- An internationalized source program file (prefixed `nl_`) that is generated during the internationalization process
- An intermediate file (suffixed `.msg`) that is created in your directory and that can be referenced by other utilities
- A message text source file containing the extracted and translated text strings (suffixed `.msf`) generated during the internationalization process

The pattern file is used to determine which strings are matched for the program being internationalized. This system-wide file is used by the extraction tools. Pattern files are described on the `patterns`(5int) reference page and in the file `/usr/lib/intln/patterns`.

The ignore file is used to instruct the string extraction tools to ignore specific strings in the source file. Each line in the ignore file contains a single string which is compared against the strings matched by the pattern file.

The format of the message text source file is described in Section 2.3. The use of the `gencat` tool is described in Section 2.4 and the `gencat`(1int) reference page.

The string extraction tools produce these files:

- An internationalized program source file that has had the text strings removed and replaced with calls to a message catalog access routine
- A message text source file, containing the text strings removed from the original program source file, for use as input to `gencat` after translation of the text

2.3 Format of the Message Text Source File

This section describes the format of a message text source file. Message text strings can be specified using either message numbers or mnemonics. Note that the fields of a message text source line are separated by a single ASCII space or tab character. Any other ASCII spaces or tabs are considered to be part of the subsequent field.

2.3.1 Set and Message Numbers

Message catalogs can be divided into one or more sets of program messages that are grouped together by a set number. The set number is a parameter of `catgets` and `catgetmsg`.

You specify the set number of following messages until the next `$set`, `$delset`, or end-of-file, by using the construct:

```
$set n comment
```

The `n` denotes the set number which must be presented in ascending order within a single source file but need not be contiguous. Any string following the set number is treated as a comment. There must be at least one `$set` directive in a message text source file before any messages.

If you are using message numbers (numeric format), you delete the entire message set from an existing message catalog using the construct:

```
$delset n comment
```

Any string following the set number is treated as a comment.

To place comments in the message text source file, type a line beginning with a dollar symbol (`$`) followed by an ASCII space or tab character and then the comment:

```
$  comment
```

To define message numbers, use the construct:

```
m  message-text
```

The `message-text` is stored in the message catalog with message number `m` and the set number specified by the last `$set` directive. If the `message-text` is empty, and an ASCII space or tab field separator is present, a null string is stored in the message catalog.

Note that `catgets` and `catgetmsg` do not distinguish between a null message and an undefined message; in both cases these routines return a pointer to the null string. Message numbers within a single set need not be contiguous, although they must be in ascending order. The length of `message-text` must not exceed the number of characters specified in the `nl_textmax` field of the file `/usr/include/limits.h`.

You can use an optional quote character `c` to surround `message-text` so that trailing spaces are visible in a message source line. You specify this by:

```
$quote  c
```

By default, or if an empty `$quote` directive is supplied, no quoting of `message-text` is recognized.

If a quote character is defined, all white space between the message number and the quote is ignored. Empty lines in a message text source file are always ignored.

Text strings can contain the special characters and escape sequences. Escape sequences recognized by `gencat` are defined in Table 2-1.

Table 2-1: Escape Sequences Recognized by gencat

Description	Symbol	Sequence
newline	NL(LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
octal value	ddd	\ddd

The escape sequence `\ddd` consists of a backslash followed by 1, 2, or 3 octal digits which specify the value of the desired character. If the character following a backslash is not one of those specified, the backslash is ignored. You also use a backslash to continue a string on the following line. Thus, the following two lines describe a single message string:

```
1 This line continues \
to the next line
```

which is equivalent to:

```
1 This line continues to the next line
```

The backslash must be the last character on the line that is to be continued.

Further localization is provided by translating the strings contained in the message text source file into the required languages, and by using `gencat` to create the various language message catalogs.

2.3.2 Mnemonics

Sets and messages can be given mnemonic names as an alternative to set and message numbers. A mnemonic is defined as any string starting with an alphabetic character. You cannot use mnemonics together with set and message numbers in the same source file.

In the following example, the mnemonic SET_1, HELLO and BYE are used instead of the numbers 1, 1 and 2 respectively:

```
$set SET_1
HELLO Hello world
BYE Goodbye world
```

The call

```
catgets (catd, SET_1, HELLO, "")
```

would return the message:

```
Hello world
```

The `-h` flag of the `gencat` tool forces the generation of a header file containing `#define` statements. You must include `#define` statements in the program source files when you use mnemonics. Using the previous example as a basis, the following code fragments compare two programs, one using mnemonics and the other using message numbers:

- Using mnemonics:

  ```
  #include "prog.h"
   ....
   ....
   ....
  catgets(catd, SET_1, HELLO, "Hello");
   ....
  ```

- Using message numbers:

  ```
   ....
   ....
   ....
   ....
  catgets(catd, 1, 1, "Hello");
   ....
  ```

The contents of the `.msf` message file used by the mnemonic program is of the form:

```
$quote "
$set SET_1
HELLO "Hello world"
 ....
 ....
```

Note, only the text within the quotes should be translated.

The header file generated using `gencat -h` contains the following:

```
#define SET_1 1
#define HELLO 1
#define BYE   2
 ....
```

In all other respects, the use of mnemonics does not change how the internationalization tools are used.

There are some restrictions on the use of mnemonics:

- Set and message mnemonics cannot have the same name.
- Catalogs cannot be merged using `gencat`. An old catalog is always overwritten by the new catalog.
- Mnemonics and set and message numbers cannot be combined in the same source file.

2.4 Using gencat

The `gencat` program takes a message text source file and either produces a new message catalog or merges the new message text into an existing message catalog.

- If the message catalog has already been created, and set and message numbers are being used, `gencat` merges the set and message numbers with the existing message catalog.
- If the message catalog does not exist, `gencat` creates it.

If a message text source file uses mnemonics, `gencat` does not merge the files. The new file overwrites the original file.

Set and message numbers are described in Section 2.3.1, and mnemonics are described in Section 2.3.2.

An example of the use of `gencat` is:

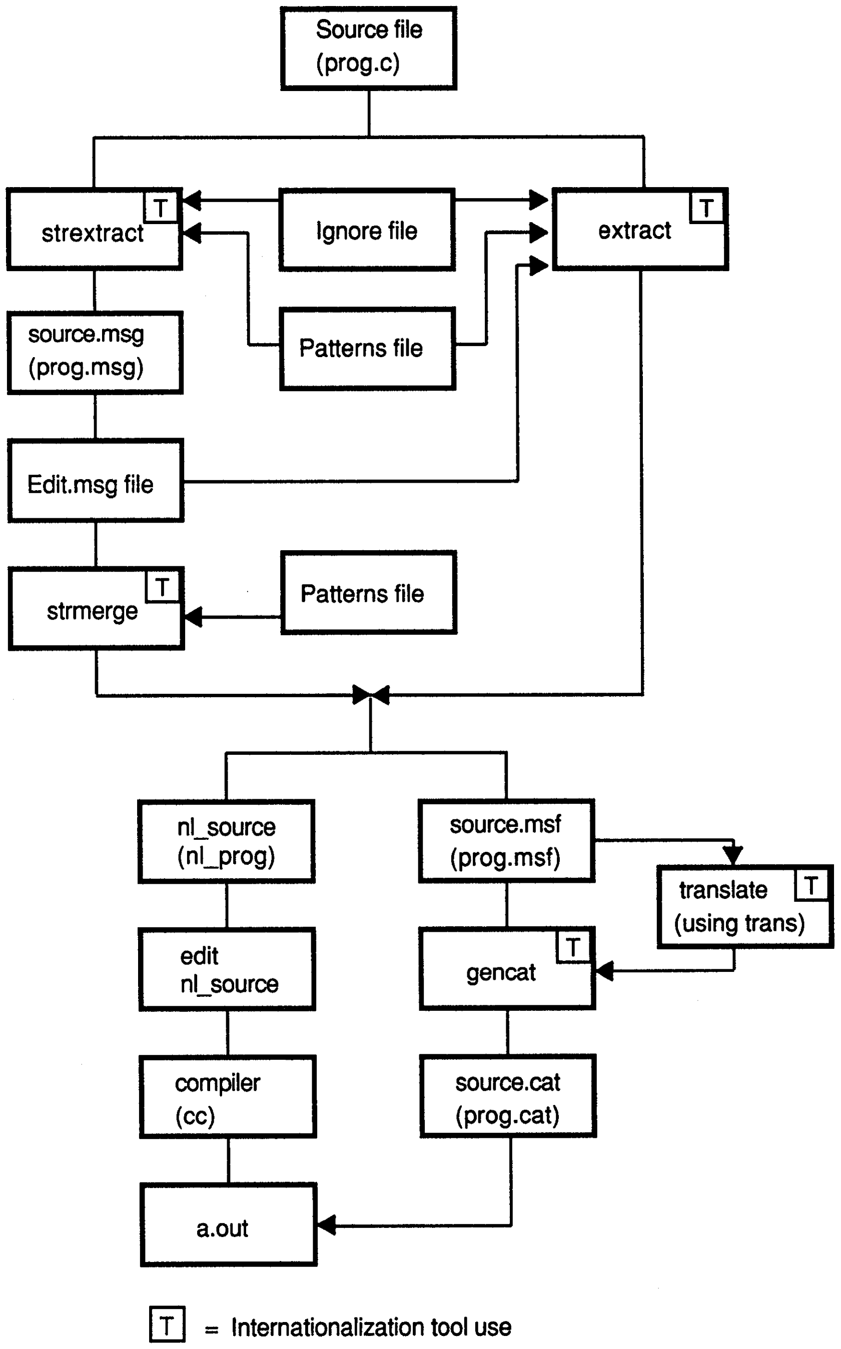
```
gencat catfile msgfile
```

where `catfile` is the name of the target message catalog and `msgfile` is the name of a message text source file. If `catfile` exists, then the messages and sets defined in `msgfile` are added to `catfile`. If set and message numbers collide, the new message text given in `msgfile` replaces the existing message text contained in `catfile`. If `catfile` does not exist, `gencat` creates it.

The software developer uses the `gencat -h` to produce the header file defining the mapping between the mnemonic message identifiers and the numbers required by `catgets` and `catgetmsg`.

The sequence of operations needed to create an internationalized source file and a translated message catalog is shown in Figure 2-1.

Figure 2-1: Creating a Message Catalog



ZK-0045U-R

The C program (prog.c) is changed into an internationalized source program (nl_prog) with the text strings removed and replaced with calls to the message catalog retrieval routines. This is done by using either the interactive extraction tool `extract`, or by using the batch extraction tool `strextract` followed by the batch merging tool `strmerge`.

The message text source file produced (prog.msf) is translated using the translation tool `trans`. A message catalog containing the translated messages (prog.cat) is then produced using the `gencat` tool.

2.5 Library Routines

This section describes the library routines used to open and close message catalogs and to extract information from within an open catalog. The library routines are as follows:

- `catopen`
- `catclose`
- `catgets`

To compile a C program, use the `-li` option to include the internationalization library, as shown in the following example:

```
cc -o prog prog.c -li
```

2.5.1 Using catopen

Message catalogs are opened for use by calling the library routine `catopen`, which locates the identified message catalog according to the search and naming rules defined in the environment variable `NLSPATH`. Refer to `environ`(5int) for details of this environment variable. The following shows an example of calling the `catopen` routine:

```
catd = catopen(argv[0], 0);
```

If successful, `catopen` returns a catalog-descriptor of type `nl_catd` which is used on subsequent calls to `catgets` and `catgetmsg` to identify the prepared message catalog. Message catalogs are closed by calling the library routine `catclose`.

2.5.2 Using catgets

The routine `catgets` retrieves a numbered message from a numbered message set in the message catalog identified by the `catd` argument. The following shows an example of calling the `catgets` routine:

```
char *catgets (catd, set_num, msg_num, s)
```

In this example, the `set_num` argument is the number of the message set containing the message `msg_num`, and `s` is a pointer to the default message string. If `catgets` retrieves the message successfully, it returns a pointer to the message text to the caller. If the call is unsuccessful because the message catalog identified by `catd` is unavailable, then `catgets` returns `s`. If `msg_num` is not contained in the message catalog identified by `catd`, `catgets` returns the null string.

All buffer handling and allocation of storage space (for holding the text of a program message) is performed internally by `catgets`. For example, the following C source program uses `catopen` and `catgets` to retrieve messages from the message catalog identified as `prog`:

```
#include <stdio.h>
#include <nl_types.h>
#define NL_SETN 1

main ()
{
    nl_catd catd = catopen ("prog", 0);
    printf ("%s\n", catgets (catd, NL_SETN, 1, "hello world"));
    catclose (catd);
}
```

Default message strings enable the text for one language to be kept with the program as an aid to readability. Alternatively, they can be used to allow application programs to continue working predictably when specific localizations of the message text are unavailable. For example, if the above program were invoked from the c shell as follows:

```
$ setenv LANG FRE_FR.8859
$ prog
```

and assuming that the French message text for `prog` was undefined on the system, then the above invocation of `prog` would cause the default message string to be displayed:

```
hello world
```

2.6 Using trans

The translation tool, `trans`, assists in the translation of source message catalogs. The command reads input from *file.msf* and writes its output either to a file named `trans.msf` or to a file you name on the command line. The command displays *file.msf* in a multiple window screen that lets you simultaneously see the original message, the translated text you enter, and any messages from the `trans` command.

A full description of the `trans` tool and the associated editor is contained on the `trans`(1int) reference page.

Message catalogs can also be translated using a standard text editor.

Program Localization 3

This chapter discusses the following topics:

- The announcement mechanism, which announces the language and cultural requirements of the program to the system
- The announcement categories
- How to set the program locale
- How to set categories to the default defined for the implementation

An internationalized program localizes its run-time behavior for a particular language, territory, and codeset by establishing the required localization data in the program's locale. You establish the localization data by calling the `setlocale` library routine, as shown:

```
setlocale (category, locale)
```

The *category* argument is a constant defined in `<locale.h>`. The following shows possible values for *category*:

LC_ALL	Affects all of the following categories
LC_COLLATE	Affects the behavior of the string collation library routines `strcoll`(3) and `strxfrm`(3)
LC_CTYPE	Affects the behavior of the character-handling library routines `conv`(3) and `ctype`(3)
LC_NUMERIC	Affects the radix and thousands separator character in the formatted input/output library routines `printf`(3int) and `scanf`(3int). LC_NUMERIC also affects the conversion library routines `atof`(3) and `ecvt`(3)
LC_TIME	Affects the behavior of the time library routine `strftime`(3)
LC_MONETARY	Affects the currency string in the library routine `nl_langinfo`(3int)

The *locale* argument is a pointer to a character string containing the required setting of *category* in the following format:

```
language[_territory[.codeset]][@modifier]
```

You can define *language*, *territory*, and *codeset* for all settings of *category*, and you can define an *@modifier* for all categories except LC_ALL.

The following preset values of *locale* are defined for all settings of *category*:

''C''	Specifies the standard environment for the C language. If `setlocale` is not invoked, the C locale is the default.

""	Specifies that the setting of the *locale* is obtained from the corresponding environment variables. Obtaining the *locale* setting from environment variables is fully explained in Section 3.5.
NULL	Directs `setlocale` to query *category* and return the current setting of *locale*. You can use the string `setlocale` returns only as input to subsequent `setlocale` calls.

To use `setlocale` to obtain the *locale* for all categories from environment variables, do the following:

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

You can also define a *locale* setting for a specific category. To define a specific category, you pass the *locale* setting directly in the `setlocale` call, as shown:

```
setlocale (LC_COLLATE, "FRE_FR.MCS")
```

This example specifies collation appropriate for the Digital Multinational Character Set (MCS) in France.

If you need to define a category more precisely than is possible using *language*, *territory*, and *codeset*, you can use *@modifier*. The following example shows a category definition that uses *@modifier*:

```
setlocale (LC_COLLATE, "FRE_FR.8859@CCOLL")
```

In this example collating is done according to the collation table, `CCOLL`, defined in the FRE_FR.8859 database, rather than the default collation table.

Preferably, you can obtain the *locale* for the LC_COLLATE category from the corresponding environment variable as follows:

```
setlocale (LC_COLLATE, "")
```

3.1 The Announcement Mechanism

When an internationalized program is run, the language requirements of the program must be announced to the system.

You define the environment variable ${LANG} to identify which *language*, *territory*, *codeset*, and *modifier* a program requires. You can define a unique value of ${LANG} for each supported *language*, *territory*, *codeset*, and *modifier* combination. If you define ${LANG} settings for different *language*, *territory*, *codeset*, and *modifier* settings, each definition might be associated with a different instance of collating sequence, character conversion, character classification, `langinfo` tables, and message catalogs.

The ${LANG} variable contains the required language, territory, codeset, and modifier names in English as follows:

```
language[_territory[.codeset][@modifier]
```

The length of the entire string should not exceed the value of `NL_LANGMAX` located in `/usr/include/limits.h`. The set of characters, excluding separators, is restricted to the ASCII set of alphanumeric characters. Language support databases and naming conventions are shown in the `lang`(5int) reference page.

On its own, *language* selects the required native language. You can specify *_territory* or *_territory.codset* if you need to be more specific than native language. The following examples demonstrate defining the LANG variable:

- Example 1

  ```
  LANG=FRE
  ```

  This example selects a database that supports the French native language.

- Example 2

  ```
  LANG=FRE_FR
  ```

  This example selects a database that supports the French native language, as it is spoken in France (rather than Canada).

- Example 3

  ```
  LANG=FRE_FR.MCS
  ```

  This example selects a database that supports the French native language, as spoken in France, and the Digital MCS. You cannot specify the Digital MCS unless you specify a *_territory*, in this case "_FR."

If the files `FRE` and `FRE_FR` are linked to the FRE_FR.MCS database, Example 1, Example 2, and Example 3 refer to the same database.

For information on creating a language support database, see Chapter 4.

3.2 Announcement Categories

The general announcement mechanism by which users can identify overall requirements for program localization is provided by the environment variable ${LANG}. This is sufficient when a single localization covers the user's requirements for text collation, character classification, and message presentation.

Selective modification of the international environment can be achieved by defining additional environment variables, one for each permitted setting of *category*, except LC_ALL. (For more information, see the `setlocale`(3) reference page.) The permitted categories are: LC_COLLATE, LC_CTYPE, LC_NUMERIC, LC_TIME and LC_MONETARY. If any of these are not defined in the current environment, `LANG` provides the necessary defaults.

LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, and LC_TIME are also defined to accept an additional field, `@modifier`, which enables you to select a specific instance of localization data within a single category (for example, for selecting dictionary-ordering of data as opposed to character-ordering of data). For example, if you want to interact with the system in French, but are required to sort German text files, you could define LANG and LC_COLLATE as follows:

```
LANG=Fr_FR
LC_COLLATE=De_DE
```

You could extend this definition to select, for example, dictionary ordering by using the `@modifier` field, as follows:

```
LC_COLLATE=De_DE@dict
```

3.3 Setting the Program Locale

There are three ways to set the program locale using the `setlocale` library routine:

`setlocale` (*category*, *string*)

This usage sets a specific *category* in the program locale to a specific value of *string*, for example;

```
setlocale (LC_ALL, "FRE_FR.MCS");
```

In this example, all categories of the program locale are set to the locale corresponding to the string FRE_FR.MCS, or the French language as spoken in France, using the Digital MCS. The string FRE_FR.MCS is used to locate the appropriate database. For more information, refer to `lang`(5int) reference page.

If *string* does not correspond to a valid setting of *locale*, `setlocale` returns a null pointer and the program locale is not changed. Otherwise, `setlocale` returns the name of the locale.

`setlocale` (*category*, *"C"*)

This usage resets the default environment for the C language.

`setlocale` (*category*, *" "*)

This usage sets *category* to correspond to the setting of the associated environment variable and is described in Sections 3.4 and 3.5.

By default, the directory `/usr/lib/intln` contains the language support databases. If you intend to place your language support databases in another directory, you specify the directory path with the INTLINFO environment variable.

3.4 Setting a Specific Category

This use of `setlocale` allows one of either LC_COLLATE, LC_CTYPE, LC_NUMERIC, LC_TIME or LC_MONETARY to be set individually. For example:

```
setlocale (LC_COLLATE, "");
```

Here, `setlocale` first checks the value of the corresponding environment variable, ${LC_COLLATE}. If the value contains the name of a valid locale, `setlocale` sets the specified category to that value and returns its name. If the value is invalid, `setlocale` returns a null pointer and the program locale is not changed.

If the environment variable corresponding to *category* is not set or is the empty string, `setlocale` examines ${LANG}. If ${LANG} is set and contains the name of a valid locale, that value is used to set *category*. Otherwise, `setlocale` returns a null pointer and the program locale is not changed.

On ULTRIX, the implementation defined default is the C locale.

3.5 Setting all Categories

This use of `setlocale` is similar to that described in Section 3.4, except that here `setlocale` examines all the environment variables to determine what values to set. In this case, `setlocale` is called as follows:

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

Here, `setlocale` first checks all the environment variables. If they are valid, `setlocale` initializes each *category* to the value of the corresponding environment variable. If any environment variable is invalid, `setlocale` returns a null pointer and the program locale is not changed.

Categories are initialized in the following order, where ${LANG} is used to initialize category LC_ALL:

1. LC_ALL
2. LC_CTYPE
3. LC_COLLATE
4. LC_TIME
5. LC_NUMERIC
6. LC_MONETARY

Using this scheme, environment variables corresponding to specific categories override the setting of ${LANG}.

If a category-specific environment variable is not set, or is set to the empty string, that category is not overwritten (that is, it assumes the setting of ${LANG}). If ${LANG} is not set, or is set to the empty string, `setlocale` returns a null pointer and the program locale is not changed. This is the default.

On ULTRIX, the implementation defined default is the C locale.

3.6 The C Locale

In the C locale, all characters are encoded in 7 bit ASCII. Also, characters are collated in machine order. The C locale is guaranteed to exist on all X/Open and POSIX compliant systems. Table 4-3 shows how national language strings are returned in the C locale.

3.7 Internationalized Program Example

The following is an example of an internationalized C program. This program, `idate.c`, displays date and time for a specified locale. The associated header and message files are shown following the source program.

```
/*
* idate: display date and time in locale specific format
*
* Sample internationalized application. This program uses the *
* mnemonic format for message catalogs to enhance maintainability *
*/

#include <sys/time.h>

#include <langinfo.h>   /* default strings for date/time *
*                          formats, etc.  */
#include <locale.h>     /* declarations used by setlocale */
#include <nl_types.h>   /* declarations for message catalog system */

#include "idate.h"      /* generated by gencat, contains message *
*                          identifiers */

nl_catd catd;
```

```
struct timeval  tp;
struct timezone tpz;

main(argc, argv)
int argc;
char *argv[];
{
   char    timestring[50];
   struct  tm *tms;

   /* open message catalog - look in current directory */

   catd = catopen("idate.cat", 0);

   /* check command line arguments */

   if (argc > 1) {
       printf(catgets(catd, IDATE_SET1, USE_MSG, "usage: incorrect\n"));
       exit(1);
   }

 /* initialize runtime locale */

 if (setlocale(LC_TIME, "") == (char *)0) {

   printf(catgets(catd, IDATE_SET1, LOCALE_MSG, "idate: cannot change \
   locale - check environment variables\n"));
 }

   /* get time from system clock */

   time(&tp.tv_sec);
   tms = localtime(&tp.tv_sec);

   /* do I18N conversion */

   strftime(timestring, sizeof(timestring), nl_langinfo(D_T_FMT), tms);

   printf("%s %s\n", catgets(catd, IDATE_SET1, TIME_MSG, \
   "Local time: "), timestring);

   /* close message catalog */

   catclose(catd);
}
```

The following is the contents of the header file for `idate`:

```
/*
 * idate.h: header file created by gencat -h idate.h
 * idate.cat idate.msf
 */
#define IDATE_SET1      0       /* set name */
#define USE_MSG 0
#define LOCALE_MSG      1
#define TIME_MSG:       2
```

The following shows the contents of the message file `idate.msf` that is used in conjunction with `idate.c`:

```
$ idate.msf

$ This is the sample message file for use with the program
$ idate.c. Note the syntax of each line with a directive.

$ Note also that blank lines are accepted as input

$ When using mnemonic format for messages you are required
$ to use a quote character and to quote each message string.

$ This file can be used as input to the trans utility.
$ trans provides a simple user interface to aid the
$ process of message text translation.

$quote "

$set IDATE_SET1
USE_MSG "usage: idate\n"
LOCALE_MSG "idate: cannot change locale, check environment variables\n"
TIME_MSG: "Local Time: "

$ End of idate.msf
```

Language Support Databases 4

The language support databases are used to hold various language dependent entities, and to free programs from national language dependencies. There is one language support database for each national language used on the system. The information in the language support databases is supplied through database language source files which enable the national language and codeset characteristics to be defined. The file comprises definitions for the following:

- Codeset
- Property table
- Collation table
- String tables
- Conversion tables

The international compiler converts these tables into an efficient binary representation suitable for use by run-time functions. The international compiler is described on the `ic(1int)` reference page.

The following general considerations apply to the database language source file:

- The database source should only contain ASCII characters.
- The source is free format, so ''white space'' has no significance other than as a separator for tokens in the input.
- You can use C-style comments and macro definitions, in particular the `#include` and `#define` facilities.

By default, the language support database files are held under `/usr/lib/intln`. The source language and the format of the source files is illustrated in Appendix B.

Example 4-1 shows the basic structure of the source file. All definitions are terminated with the ''END.'' sequence.

Example 4-1: Structure of the Source File

```
CODESET ENG_GB.MCS :
        /*
         * codeset definition and default property table
         */
END.
COLLATION :
        /*
         * default collation table
         */
END.
STRINGTABLE :
        /*
```

Example 4-1: (continued)

```
         * default string table
         */
END.
CONVERSION toupper :
        /*
         * lowercase to uppercase conversion table
         */
END.
CONVERSION tolower :
        /*
         * uppercase to lowercase conversion table
         */
END.
```

4.1 The Codeset Definition

The codeset defines the valid characters and their properties within the language. For example, it could specify that ''A'' is a valid character in the English language, possessing lowercase and hexadecimal properties.

The definition of the codeset being used starts with the keyword CODESET followed by the codeset name `double` letters. For example, ê in ISO6937 is replaced by the sequence e´.

Once compilation is successful, the name given to the codeset becomes the name of the binary file. In most cases, this name is in the following format:

```
language_[territory[.codeset][@modifier]]
```

You can specify the name of the codeset on the `ic` command line using the `-o` option. If you specify a name on the command line, the name you specify supersedes the name of the codeset in the database source file.

After the keyword assignment, each code is defined by assigning the value of the code to an identifier. This identifier can be used to reference the code from then on. This assignment has the form:

```
Identifier '=' value_list [ ':' Properties ] ';'
```

For example:

```
a = 'a' : LOWER, HEX;
```

The `value_list` is a list of values separated by commas. A value may be given as a C-style character constant (' '), in octal (0nnn), hexadecimal (0xnnn), decimal (nnn), ISO notation (mm/nn), or by giving the name of a previously defined code.

Codes may be either simple or combined. However, several restrictions must be observed when defining codes in the CODESET section:

- The list of simple codes must contain all codes from code value 0x0 up to and including the code with the highest value defined. The order of definition is not important, since all code values are sorted into ascending collation order when the whole codeset definition has been read.
- The list of simple codes may not contain codes with duplicate code values.
- There may be up to 2^{15} definitions for multi-byte codes. Combined codes need not have contiguous code values and will be sorted in ascending machine collation order and construct the ''double letter table'' in the compiled database.

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

The optional `properties` part of the definition assigns 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 require all the letters defined in a standard code set. Properties take the form of a list of keywords separated by commas.

A third kind of statement allowed in the CODESET section is the (re-)assignment of default properties to an already defined code. This statement takes the form of

```
Identifier ':' Properties ';'
```

The use of the `#include` facility provided in the language is strongly recommended as most of the codes considered contain common code (for example ASCII or ISO646) in their lower half. Using a common `include` file reduces the risk of error and provides a common name basis for the remainder of the source.

4.2 The Property Table

The property table contains the mapping between characters in the codeset and classification. Each character code from the coded character set is used to index an entry in the relevant language property table. Each entry in the property table contains a series of flags identifying whether a particular language assertion is true or false. The character may possess any of the following attributes:

- Undefined
- Uppercase alphabetic
- Lowercase alphabetic
- Punctuation
- Control
- Blank

These can be accessed at run time by the `ctype` library routines.

There can be more than one property table. Each property table is introduced by the keyword PROPERTY. The default property table, built along with the code set, has the predefined name PROP_DFLT. The property table must not be redefined. Names of property tables must be unique throughout the source.

A statement in the property table takes the form of:

```
Identifier ':' Properties ';'
```

where `Identifier` designates a defined code and `Properties` is a list of properties separated by commas. For example:

```
C: UPPER, HEX;
```

Some properties effect the interpretation of characters by various other internationalization library routines. For example, the property DIPHTONG must be set for diphthongs to collate correctly as diphthongs, and the property DOUBLE must be set to recognize correctly the first of a double-letter sequence.

The full list of properties is shown in Table 4-1.

Table 4-1: Properties and Character Classification

Property	Character Classification
ARITH	arithmetic sign
BLANK	blank character
CTRL	control character
CURENCY	currency character
DIACRIT	diacritical sign
DIPHTONG	diphthong
DOUBLE	double letter
FRACTION	fraction character
ILLEGAL	illegal character
LOWER	lowercase letter
MISCEL	miscellaneous symbol
PUNCT	punctuation character
SPACE	space character
SUPSUB	superscript or subscript
UPPER	uppercase letter

The corresponding code to the property DOUBLE is constructed from two other single-byte codes, but it is treated as a single code. This treatment allows the following:

- The expansion of 8-bit character sets to allow double letters (for example Ll or ll in Spanish) that collate two-to-one
- The handling of 8/16 bit codes like ISO6937/1, which is the character "ê"

The corresponding code to the property DIACRI, for example, is a diacritical sign. If combined with either UPPER or LOWER, the corresponding code is a diacritical letter.

The meaning of diphthong in internationalization is somewhat different from the definition used in the grammar of languages that use diphthongs. Diphthong, for the purposes of internationalization, is defined as a character for which one-to-two collation must be used. This implies an interdependence with the collation tables.

The properties of a code can be redefined by the user since only the definition in effect upon reaching the end of the property table will be put in the binary file.

A code with no defined property will be listed as ILLEGAL in the resulting property table.

4.3 The Collation Table

Collation tables define the collating sequence for each supported language. The binary values of characters in the associated coded character-set are used as indices into the table. Individual entries are used to indicate the relative position of that character in the language collating sequence. The package supports the following:

- One-to-one character mappings, such that ''a'' collates before ''b,'' and so on.
- One-to-two character mappings, where certain characters are treated as two characters. For example, the German sharp ''s,'' becomes ''ss'' for collating.
- Two-to-one character mappings, where certain character sequences are treated as a single character in the collating sequence. For example, ''ch'' and ''ll'' in Spanish are collated after ''c'' and ''l'' respectively.
- No preference characters, where certain characters are ignored by the collating sequence. For example, if ''-'' is defined as a no preference character, then the strings ''re-locate'' and ''relocate'' are equal.

These capabilities provide support for collating algorithms which cater for case and accent priority, where for example, two characters are first compared for equality, ignoring accents, and if equal are then ordered by accent sequence. Collating algorithms of this type gives a dictionary ordering of data. The dictionary ordering of data within the internationalization package is the same as for a normal dictionary in the language being considered. Telephone book ordering is the same as for a telephone directory in the supported language. It should be noted that both dictionary and telephone book ordering may be subject to local variation.

The default collation table is introduced by the keyword COLLATION, and is named COLL_DFLT. The default table must exist for `ic` to compile the database. Other collation tables can be introduced by the keyword COLLATION, followed by the name of the table. Names of collation tables must be unique throughout the source.

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

- PRIMARY ':' Ident_list ';' for example,

  ```
  PRIMARY: a, A, b, B;
  ```

- PRIMARY ':' Ident '-' Ident ';' for example,

  ```
  PRIMARY: a-z;
  ```

- PRIMARY ':' REST ';' for example,

  ```
  PRIMARY: REST;
  ```

- EQUAL ':' Ident_list ';' for example,

  ```
  EQUAL: a,A;
  ```

- Ident '=' '(' Ident ',' Ident ')' ';' for example,

  ```
  PRIMARY: ae = (a, e);
  ```

- PROPERTY ':' Property_table_name ';' for example,

  ```
  PROPERTY: newprop;
  ```

The order of statements in the collation section is significant. All of the statements (except the last) open a new class of codes with primary and secondary weights. The primary weight is set by the position of the PRIMARY or EQUAL statement, with all the codes named in the statement having the same primary weight. For example, the sixth PRIMARY statement in a collation section would assign the primary weight 6 to all the codes listed. Primary weights start at 1 and increase by one for each statement encountered up to a limit of 254. The secondary weight of the codes is governed by their ordering within a set, except codes with an EQUAL statement, which all have the same secondary weight. The limit on secondary weights is 255.

The statement PRIMARY ':' Ident_list ';' assigns the named codes ascending secondary weights from left to right.

The statement PRIMARY ':' Ident '-' Ident ';' assigns ascending secondary weights for ascending machine collation order to the named codes.

The statement PRIMARY ':' REST ';' sets the primary weight of codes not explicitly named in the collation section. The secondary weight of the codes is set to ascending machine collation order. This is a convenient notation for defaulting unspecified codes to collate after or before all others.

The statement EQUAL ':' Ident_list assigns the same PRIMARY and SECONDARY weight to all codes in the list.

The statement Ident '=' '(' Ident ',' Ident ')' ';' is reserved for the collation of diphthongs (one-to-two collation). It implies that the left hand side code collates as if it were the first right hand code followed by the second right hand code.

In order for the diphthong collation to work correctly, the code named on the left hand of the statement must be marked as DIPHTONG in at least one property table. If this property table is not the default table, the statement PROPERTY ':' Property_table_name ';' must be used to identify the property table name to the compiler. This allows the run-time routines to load a collation-only property table for use with diphthongs.

Table 4-2 gives three examples of primary and secondary weighting. In Example 1, all the items have the same primary weight, but have ascending secondary weights. In Example 2, both primary and secondary weights are used to resolve collation. In Example 3, all the items have the same secondary weight, but have ascending primary weights.

If the three alphabetic strings:

- Abc
- aac
- Bbc

were collated using the three examples in Table 4-2, the results would be as follows:

- Example 1: Abc, aac, Bbc
- Example 2: aac, Abc, Bbc
- Example 3: Abc, aac, Bbc

Note that Example 2 is the only way to obtain dictionary collation. Of Examples 1 and 3, Example 3 is the most efficient since only one pass is required. Collation is resolved on primary weighting, then secondary weighting.

Table 4-2: Examples of Primary and Secondary Weighting

Example 1							
secondary		1	2	3	4	5	6
primary	1	A	a	B	b	C	c

Example 2			
secondary		1	2
primary	1	A	a
	2	B	b
	3	C	c

Example 3		
secondary		1
primary	1	A
	2	a
	3	B
	4	b
	5	C
	6	c

If a code is not given weights in the collation section, it is treated as having the (otherwise illegal) primary and secondary weight 0 (zero). This results in the code collating as a ''don't care'' character.

Double letters (2-to-1 collation) must be named in the codeset. They can then be given a weight in the collation section.

For some examples on collation sequences, refer to Appendix B.

4.4 The String Table

The string table contains the language strings required for formatting date and time, yes and no, and radix characters. The default string table is introduced by the keyword STRINGTABLE, and is named STRG_DFLT. The default string table must exist for `ic` to compile the database. Other string tables can be introduced by the keyword STRINGTABLE, followed by the table name. However, names of string tables must be unique throughout the source.

Each statement in a string table has the form:

```
Ident '=' value_list ';'
```

where `Ident` is an identifier, the name of the string and `value_list` is a comma separated list of strings, character constants, and identifiers designating codes. This allows inclusion of non-ASCII codes in any string table by giving the name of the code in `value_list`.

Table 4-3 shows the strings that must appear in the string table.

Table 4-3: Mandatory Strings in the String Table

String	Meaning	C locale	Category
NOSTR	Negative response	no	LC_ALL
YESSTR	Positive response	yes	LC_ALL
D_T_FMT	Default date and time format	%a %b %d %H:%M:%S %Y	LC_TIME
D_FMT	Default date format	%m/%d/%y	LC_TIME
T_FMT	Default time format	%H:%M:%S	LC_TIME
DAY_1	Day name	Sunday	LC_TIME
DAY_2	Day name	Monday	LC_TIME
....	....	....	....
DAY_7	Day name	Saturday	LC_TIME
ABDAY_1	Abbreviated day name	Sun	LC_TIME
ABDAY_2	Abbreviated day name	Mon	LC_TIME
ABDAY_3	Abbreviated day name	Tue	LC_TIME
....	....	....	....
ABDAY_7	Abbreviated day name	Sat	LC_TIME
MON_1	Month name	January	LC_TIME
MON_2	Month name	February	LC_TIME
MON_3	Month name	March	LC_TIME
....	....	....	....
MON_12	Month name	December	LC_TIME
ABMON_1	Abbreviated month name	Jan	LC_TIME
ABMON_2	Abbreviated month name	Feb	LC_TIME
....	....	....	....
ABMON_12	Abbreviated month name	Dec	LC_TIME
RADIXCHAR	Radix character		LC_NUMERIC
THOUSEP	Thousands separator		LC_NUMERIC
CRNCYSTR	Currency format		LC_MONETARY
AM_STR	String for AM	AM	LC_TIME
PM_STR	String for PM	PM	LC_TIME
EXPL_STR	Lowercase exponent character	e	LC_NUMERIC
EXPU_STR	Uppercase exponent character	E	LC_NUMERIC

4.5 The Conversion Tables

The conversion tables are used to convert characters within the codeset, for example uppercase converted to lowercase. There must be at least two conversion tables within the database language source file. These are named *toupper* and *tolower* and are used to convert characters to uppercase and to lowercase respectively.

A statement in a conversion table takes one of three forms in which `Ident` specifies a code defined in the codeset, and `conversion_value` specifies the code or string value that the left hand side should be converted to.

- Ident '->' conversion_value ';'
  For example: `a -> A;`
- Ident '-' Ident '->' Ident '-' Ident ';'
  For example: `a-z -> A-Z;`
- DEFAULT '->' default_value ';'
  For example: `DEFAULT -> SAME;`

The default value for a conversion may be given using the `DEFAULT` statement. Any code without a specified conversion, maps to the given value. There are two predefined values possible in a `DEFAULT` statement:

- VOID, which means that all other codes convert to either the ASCII NUL code (in the case of a code conversion) or to an empty 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. This default conversion is not valid for string type conversions.

The range notation in the conversion section implies an underlying machine collation sequence and 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 ;
```

Some examples of both types of conversion are given in Appendix B.

Database Source Language Syntax Description A

This appendix describes the database source language you use to create a source file for a language support database. The appendix explains the syntax elements of the source files and gives an Extended Backus-Naur Form (EBNF) notation of the syntax recognized by the `ic` compiler.

A.1 Rules for Building Identifiers

The rules for building an identifier (Ident) are as follows:

- Each identifier must start with a letter or a hyphen (-).
- An identifier can be any length and can contain letters (a to z and A to Z), digits (1 - 9), hyphens (-), and periods (.).
- If you use a period in an identifier, at least one letter, digit, or hyphen must follow the period.

A.2 Rules for Building Strings

The rules for building a string (String) are as follows:

- No string can contain more than 255 characters.
- Each string must be enclosed in quotation marks (" ").
- Each string must be on one line in the source file.
- A string can contain the following escape sequences:

  \n — ASCII newline
  \r — ASCII return
  \t — ASCII tabulator
  \b — ASCII backspace
  \f — ASCII form feed
  \\ — escaped backslash
  \" — escaped double quotes

A.3 Rules for Building Constants

A constant (Constant) can be any of the following forms:

- A character constant, such as one character enclosed in single quotation marks (' '). You can use escape sequences within a character constant by following the C language rules for using escape sequences. For information on those rules, see the *Guide to VAX C*.

- A hexadecimal constant of the form 0x*nnnn*, where *n* designates a hexadecimal digit (0-9, a to f, and A to F). The hexadecimal constant must be in the range of 0 to 0x7FFF. You can omit leading null valued digits.
- An octal constant of the form 0*nnnn*, where *n* designates an octal digit (0-7). The octal constant must be in the range of 0 to 077777. You can omit leading null valued digits.
- A character in ISO notation *n*/*n*, where *n* designates a decimal number in the range of 0 to 15.
- A decimal number *n*, where *n* is a positive integer in the range 0 to 32,767.

A.4 Rules for Separating Tokens, Specifying Comments, and Using Directives

You must separate tokens with spaces or horizontal tabs. You must not include white space within tokens. White space (for example, " ", newline, horizontal tab) is significant only as a token separator. The `ic` compiler ignores white space that you use to make your source file readable.

As in the C language, comments are delimited by pairs of slashes and asterisks (/*comment*/). You can include comments anywhere in the source file except within tokens. If you use a comment within a token, the `ic` compiler considers the token to end where the comment begins. Any text that follows the comment begins a new token.

Because the database source file is preprocessed by the C preprocessor, you can use the preprocessor directives, such as `#include`, `#define`, and `#if`, thoughout the source file.

A.5 EBNF Description

Example A-1 contains the EBNF description of the database source language. If you are unfamiliar with EBNF notation, you can find a description of it in *Compilers, Principles, Techniques, and Tools.*[1]

The notation in this appendix differs from the description in *Compilers, Principles, Techniques, and Tools* in the following ways:

- In productions, nonterminals on the left side are separated from terminals or tokens on the right side by a colon (:) instead of an arrow.
- Terminals appear in single quotation marks (' ') or in uppercase characters, instead of boldface type.
- The nonterminals Ident, String, and Constant are not described by a production. These nonterminals are described by the rules in Section A.1, Section A.2, and Section A.3, respectively.

[1] Alfred V. Aho, *Compilers, Principles, Techniques, and Tools* (Reading, Mass: Addison-Wesley Publishing Co., 1986), pp. 26.

Example A-1: EBNF Description of the Database Source Language

```
intl_data_base
        : codeset_table data_tables

data_tables
        : data_table | data_tables data_table

data_table
        : property_table
        | collation_table
        | format_table
        | conversion_table

codeset_table
        : CODESET Ident ':' code_definition_list END '.'

code_definition_list
        : code_definition
        | code_definition_list ';' code_definition

code_definition
        : Ident '=' code_value ':' property_list
        | Ident '=' code_value
        | property_definition

code_value
        : code | code_value ',' code

code
        : Constant | Ident

property_list
        : property | property_list ',' property

property_table
        : PROPERTY Ident ':' property_definition_list END '.'

property_definition_list
        : property_definition
        | property_definition_list ';' property_definition

property_definition
        : Ident ':' property_list



property
        : ARITH | BLANK | CTRL | CURENCY | DIACRIT
        | DIPHTONG | DOUBLE | FRACTION | HEX | ILLEGAL
        | LOWER | MISCEL | NUMERAL | PUNCT
        | SPACE | SUPSUB | UPPER

collation_table
        : COLLATION ':' collation_list END '.'
        | COLLATION Ident ':' collation_list END '.'

collation_list
        : collation | collation_list ';' collation

collation
        : PRIMARY ':' code_value_list
        | PRIMARY ':' Ident '-' Ident
        | PRIMARY ':' REST
        | EQUAL ':' code_value_list
```

Example A-1: (continued)

```
        | EQUAL ':' Ident '-' Ident
        | EQUAL ':' REST
        | Ident '=' '('  Ident ',' Ident ')'
        | PROPERTY ':' Ident

code_value_list
        : Ident | code_value_list ',' Ident

format_table
        : STRINGTABLE ':' format_list END '.'
        | STRINGTABLE Ident ':' format_list END '.'

format_list
        : format | format_list ';' format

format
        : Ident '=' format_value

format_value
        : code_or_string | format_value ',' code_or_string

code_or_string
        : code | String



conversion_table
        : CONVERSION Ident ':' conversion_list END '.'
        | CODE CONVERSION Ident ':' conversion_list END '.'

conversion_list
        : conversion | conversion_list ';' conversion

conversion
        : DEFAULT '->' default_value
        | Ident '->' conversion_value
        | Ident '-' Ident '->' Ident '-' Ident

default_value
        : VOID | SAME | conversion_value

conversion_value
        : code_or_string
        | conversion_value ',' code_or_string
```

Example Source Language File B

Example B-1 illustrates the file structure of a source file for a language support database. The example omits parts of the source file to save space. To see a complete database source file, display or print one of the source files in subdirectories of the `/usr/lib/intln` directory. For example, the source file for the German database that uses the ISO Latin 1 codeset is in the `/usr/lib/intln/8859/GER_DE.8859.in` file.

Example B-1: Example of a Language Support Database Source File

```
/*
 * example annotated (partial) source for
 * a Language Support Database
 */
CODESET CH_ASCIIPLUS :
        /*  CH_ASCIIPLUS will be the name of the INTLINFO file */
#include "ISO646"
        /* include ISO646 as the predefined ASCII code definition */

        /*
         * additional definitions for demonstration purposes:
         *
         * first we have a range of secondary control codes.
         * This is not enforced by the ic compiler nor by
         * the language but is a common IS 2022 style
         * code set extension technique. Note that because
         * there are no properties defined below all these
         * codes are defined but not legal.
         */
        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;

        /*
         * NOTE: this gap in the source will prevent compilation.
         * This was done to shorten the example.
         */

        /*
         * now come some more useful code definitions. These
         * definitions are taken from the IS 8859/1
         * definition. Note the convention of writing
         * uppercase letters in all uppercase, lowercase
         * letters and special codes in all lowercase.
         * Here the codes are defined directly from their
         * ISO notation.
         */
        A_GRAVE = 12/0  : UPPER;
        A_AIGU = 12/1   : UPPER;
        A_CIRCON = 12/2 : UPPER;
        A_TILDE = 12/3  : UPPER;
        DIA_A = 12/4    : UPPER;
```

Example B-1: (continued)

```
        A_CIRCLE = 12/5 : UPPER;
        /*
         * The following declaration of AE as a diphthong enables
         * the correct treatment of diphthongs (one-to-two
         * collation) in the default collation.
         */
        AE = 12/6       : UPPER, DIPHTHONG;

        /*
         * NOTE: this gap in the source will prevent compilation.
         * This was done to shorten the example.
         */

        /*
         * lowercase equivalents of the codes defined
         * in the last block
         */
        a_grave = 14/0  : LOWER;
        a_aigu = 14/1   : LOWER;
        a_circon = 14/2 : LOWER;
        a_tilde = 14/3  : LOWER;
        dia_a = 14/4    : LOWER;
        a_circle = 14/5 : LOWER;
        ae = 14/6       : LOWER, DIPHTHONG;

        /*
         * special double letters for Spanish
         * Note that these "characters" are not defined by
         * any standard! They represent an extension
         * useful to handle the following problems:
         *      - two to one collation
         *      - conversions toupper and tolower
         */
        Ll = L, l : DOUBLE, UPPER;
        ll = l, l : DOUBLE, LOWER;

END.

/*
 * Collation table that shows most of the possible
 * problems in collation but does not make very much
 * sense in the real world:
 *
 * Uppercase and lowercase letters are intermixed and
 * within one letter the uppercase comes before the
 * lowercase letter.
 *
 * Accented characters sort after their corresponding
 * nonaccented base character.
 *
 */
COLLATION :
        PRIMARY : A, A_GRAVE, A_AIGU, A_CIRCON, A_TILDE,
                  DIA_A, A_CIRCLE;
        PRIMARY : a, a_grave, a_aigu, a_circon, a_tilde,
                  dia_a, a_circle;
        PRIMARY: B; PRIMARY: b; PRIMARY: C; PRIMARY: c;
        PRIMARY: D; PRIMARY: d; PRIMARY: E; PRIMARY: e;
        PRIMARY: F; PRIMARY: f; PRIMARY: G; PRIMARY: g;
        PRIMARY: H; PRIMARY: h; PRIMARY: I; PRIMARY: i;
        PRIMARY: J; PRIMARY: j; PRIMARY: K; PRIMARY: k;
        PRIMARY: L; PRIMARY: l;
```

Example B-1: (continued)

```
        /*
         * TWO-TO-ONE COLLATION:
         *
         * For Ll and ll Spanish collation rule says that
         * this has to be collated after L or l.
         */
        PRIMARY: Ll; PRIMARY: ll;

        PRIMARY: M; PRIMARY: m; PRIMARY: N; PRIMARY: n;

        /*
         * ONE-TO-TWO COLLATION:
         *
         * The following two codes are diphthongs, that is
         * codes that collate as two characters.
         */
        AE = (A, E);                          ae = (a, e);

        /*
         * The rest of the codes defined in the codeset will
         * collate as don't care characters.
         */

END.


/*
 * This is a sample string table based on the German language.
 *
 * Note the mixed uses of ASCII strings and identifiers
 * specified in the codeset definition.
 *
 * The strings for CRNCYSTR, D_T_FMT, D_FMT, T_FMT are
 * typically specified as ASCII strings.
 *
 * Each of the items specified is required by the ic
 * compiler. Additional items can be specified if so
 * desired.
 */


STRINGTABLE :
        NOSTR        = "nein";
        EXPL_STR     = 'e';
        EXPU_STR     = 'E';
        RADIXCHAR    = comma;
        THOUSEP      = dot;
        YESSTR       = "ja";
        CRNCYSTR     = "+DM";

        D_T_FMT      = "%a, %d. %b %Y %H:%M:%S" ;
        D_FMT        = "%a, %d. %b %Y";
        T_FMT        = "%H:%M:%S";
        AM_STR       = "AM";
        PM_STR       = "PM";

        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";
```

Example B-1: (continued)

```
        ABDAY_5      = "Do";                ABDAY_6      = "Fr";
        ABDAY_7      = "Sa";

        MON_1        = "Januar";            MON_2        = "Februar";
        MON_3        = M, dia_a, "rz";      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.


STRINGTABLE :
MON_1 = "January";
YESSTR = "oui";
END.
```

Associated Reference Pages C

This appendix gives a list of the ULTRIX reference pages associated with the Internationalization package.

iconv(1)	International codeset conversion
extract(1int)	Interactive string extract and replace
gencat(1int)	Generate a formatted message catalog
ic(1int)	Compiler for language support database
strextract(1int)	Batch string extraction
strmerge(1int)	Batch string replacement
trans(1int)	Translation tool for use with message source files
atof(3)	Convert ASCII to numbers
conv(3)	Translate characters
ctype(3)	Character classification macros
ecvt(3)	Output conversion
setlocale(3)	Set localization for internationalized program
strcoll(3)	String collation comparison
strftime(3)	Convert time and date to string
strxfrm(3)	String transformation
intro(3int)	Introduction to the internationalization subroutines
catgetmsg(3int)	Get message from a message catalog (Provided for X/Open XPG–2 conformance)
catgets(3int)	Read a program message
catopen(3int)	Open/close a message catalog
nl_langinfo(3int)	Language information
nl_printf(3int)	Print formatted output (Provided for X/Open XPG–2 conformance)
nl_scanf(3int)	Convert formatted input (Provided for X/Open XPG–2 conformance)
printf(3int)	Print formatted output
scanf(3int)	Convert formatted input
vprintf(3int)	Print formatted output of a varargs argument list
printf(3s)	Print formatted output
scanf(3s)	Convert formatted input
environ(5int)	NLS environment variables
lang(5int)	Language names
nl_types(5int)	Language support database types
patterns(5int)	Patterns for use with internationalization tools

Glossary

This glossary defines a number of technical terms that may be encountered. In some cases, the terms have not been used in the generally accepted way.

ASCII

American Standard Code for Information Interchange.
ASCII is the traditional ULTRIX coded-character set and defines 128 characters, including both control characters and graphic characters, represented by 7-bit binary values (see also ISO 646).

Character

A member of a set of elements used for the organization, control, or representation of text.

Character Set

A set of alphabetic or other characters used to construct the words and other elementary units of a national language or a computer language.

Coded Character Set

A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation.

Collating Sequence

The ordering sequence applied to characters or a group of characters when they are sorted.

Composite Graphic Symbol

A graphic symbol consisting of a combination of two or more other graphic symbols in a single character position, such as a diacritical mark and a basic letter.

Control Character

A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text.

Downshifting

The conversion of an uppercase character to its lowercase representation.

Graphic Character

A character, other than a control character, that has a visual representation when hand-written, printed, or displayed.

Internationalization

The provision within a computer program for adapting to the requirements of different national languages, local customs, and coded character sets.

ISO 646

ISO 7-bit coded character set for information interchange. The reference version of ISO 646 contains 95 graphic characters, which are identical to the graphic characters defined in the ASCII coded character set.

ISO 6937

ISO 7-bit or 8-bit coded character set for text communication using public communication networks, private communication networks, or interchange media such as magnetic tapes and discs.

ISO 8859/1

ISO 8-bit single-byte coded character set Part 1, Latin Alphabet No. 1. The ISO 8859/1 character set comprises 191 graphic characters covering the requirements of most of Western Europe.

LANG

The environment variable `LANG`, used to announce the user's requirements for national language, local customs, and coded character set to the computer system.

Local Customs

Refers to the conventions of a geographical area or territory for such things as date, time, and currency formats.

Localization

The process of establishing the run-time environment of an internationalized computer program to meet the requirements of particular national languages, local customs, and character sets.

MCS

Digital Equipment Corporation's Multinational Character Set. This is based on ISO 8859/1. It covers the requirements of most Western European languages but also includes special computer oriented symbols.

Message Catalog

A file or storage area containing program messages, command prompts, and responses to prompts for a particular national language, territory, and codeset.

National Language

A computer user's spoken or written language, such as English, French, Italian, or Spanish.

NLSPATH

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

Non-spacing Characters

A character, such as a character representing a diacritical mark in the ISO 6937 coded character set, which is used in combination with other characters to form composite graphic symbols.

Radix Character

The character that separates the integer part of a number from the fractional part.

Upshifting

The conversion of a lowercase character to its uppercase representation.

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Guide to Developing International Software
AA-LY26B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

____

____

What do you like best about this manual? ______________________________

____

____

What do you like least about this manual? ______________________________

____

____

Please list errors you have found in this manual:

Page	Description
______	______________________________
______	______________________________
______	______________________________
______	______________________________
______	______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

What version of the software described by this manual are you using? __________

Name/Title ______________________________ Dept. __________

Company ______________________________ Date __________

Mailing Address ______________________________

______________________________ Email ______________________________ Phone __________

------- Do Not Tear – Fold Here and Tape ---

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

------- Do Not Tear – Fold Here ---

Cut
Along
Dotted
Line