

# International Character Support in Allegro CL

This document contains the following sections:

## [1.0 Introduction](#)

## [2.0 Internal Representation](#)

### [2.1 History](#)

### [2.2 Unicode](#)

### [2.3 Memory Usage](#)

### [2.4 Character names](#)

## [3.0 External formats](#)

### [3.1 External-Format Overview](#)

#### [3.1.1 Basic External-Format Types](#)

#### [3.1.2 The unicode and fat External-Format Types](#)

#### [3.1.3 Composed External-Formats](#)

#### [3.1.4 Defining External-Formats](#)

#### [3.1.5 Retrieving Existing External-Formats](#)

#### [3.1.6 External-Format Runtime Mode](#)

### [3.2 External-Format Usage](#)

#### [3.2.1 Streams](#)

#### [3.2.2 String <-> External-Format Lisp Arrays](#)

### [3.3 External-Formats in 8-bit Lisp.](#)

### [3.4 Older Allegro CL External-Format Compatibility](#)

## [4.0 Foreign-Functions](#)

## [5.0 External formats and locales](#)

### [5.1 The initial locale when Allegro CL starts up](#)

### [5.2 Locales in applications](#)

## [6.0 Localization support in Allegro CL](#)

### [6.1 Introduction to locales](#)

### [6.2 Locale Definition](#)

### [6.3 Locale Attribute Accessors](#)

## [7.0 String collation with international characters](#)

## [8.0 Earlier International Allegro CL Compatibility](#)

### [8.1 EUC Module](#)

### [8.2 :mode Option Removal](#)

## [Appendix A. Functions, Symbols, Variables Documentation](#)

### [Appendix A.1. External-Format API](#)

## [Appendix B. #\newline Discussion](#)

## [Appendix C. 8-bit images](#)

## [Appendix D. Links to Unicode Reports](#)

## 1.0 Introduction

Starting with Allegro CL Release 6.0, the International Version of Allegro CL, which has been available to Unix users since Release 3.1, and to Windows users since Release 5.0.1, takes center stage to become the new standard Allegro CL. The older, non-international version remains available to Allegro CL users, as described in [Appendix C 8-bit images](#).

For most users, this version change (from 5.0.1 to 6.0 and later) will be undetectable. Some users may notice new warnings about how to improve code compatibility between non-international and International Allegro CL, especially regarding string passing to foreign functions (described below in this document). The main benefits, however, of the new version are for users developing internationalized applications requiring non-English character sets and locales outside the United States. The internal changes to Allegro CL allow for universal character representation and character Input/Output that is more flexible than that available with previous Allegro CL releases. This document describes these changes and how Lisp programmers can exploit the new features.

**Note on examples with non-ASCII characters:** This document contains some examples with non-ASCII characters to illustrate Allegro CL's ability to represent them. These examples are displayed using JPEG pictures, and therefore you cannot cut and paste them, as you can with examples containing only ASCII characters.

---

---

## 2.0 Internal Representation

---

### 2.1 History

In Allegro CL versions 5.0.1 and earlier, the standard, non-international, 8-bit version of Allegro CL represented characters internally using 8-bits per numeric character code. In Allegro CL, English letters and punctuation characters are represented using the ASCII character set. Several non-ASCII characters in 8-bit extended character sets, including the ISO-8859 series of character sets, define numeric codes for non-English/non-ASCII characters. Although non-international Allegro CL does not provide specific support to 8-bit extended character sets, all 8-bit character codes are representable in non-international Allegro CL.

The International version of Allegro CL was originally developed to support Japanese characters of which there are too many to represent using the standard 8-bit per character code model. Starting with Release 6.0, Allegro CL is extended to support all international characters (i.e., Asian, European, etc.) by using 16-bit Unicode as the

internal character representation model. The Unicode standard is used as the internal representation model for the Windows NT/2000 Operating System as well as the Java programming language.

The International version of Allegro CL has the feature `:ics` on the `*features*` list. The non-International version does not have that feature.

## 2.2 Unicode

Each character, be it a letter, Chinese ideograph, Korean Hangul, punctuation mark, or other glyph, has a unique numeric Unicode representation value. Please visit the Unicode Web site ([www.unicode.org](http://www.unicode.org)) for more information on the Unicode standard. We provide a basic description of Unicode in this document.

The characters from the Latin-1 (aka ISO 8859-1) character set, a 256 character (i.e., 8-bit) superset of the ASCII character set, have the same values in Unicode as they do in Latin-1. This provides convenient compatibility for programs which depend on numeric character codes strictly within the Latin-1 range.

Characters from other sets, however, may have different values in Unicode. For example, the Latin-2 character "Latin Capital Letter L With Stroke" has value `#xa3` in Latin-2 (ISO 8859-2), but has value `u+0141` in unicode [we use the `'u+xxxx'` convention here for describing unicode values; `'xxxx'` is in hexadecimal format]. Thus, programs which depend on character code values of non-Latin-1 characters may need to be examined and possibly updated to operate with Allegro CL. Users with existing Allegro CL programs who do not wish immediately to update their programs will be able to use the non-international, 8-bit character based, Allegro CL which does not use Unicode to represent characters.

Note that, as described later in this document, conversions from external format encodings, such as Latin-2, happen automatically during Lisp Input/Output. Thus, the only areas where user code is likely to be affected by differences between internal character representation among non-international and International Allegro CL are places where the Lisp functions `char-code` and `code-char` are called directly on non-Latin-1 characters. For example, using the Latin-2 character "Latin Capital Letter L With Stroke", the following sessions show the differences:

```
Non-international (8-bit characters) Allegro CL:
> (char-code #\l)
163          ;;; This is the (8-bit) Latin-2 code.
```

```
International Allegro CL:
> (char-code #\l)
321          ;;; This is the Unicode code.
```

## 2.3 Memory Usage

Internally, all Lisp strings are represented as arrays of Unicode character codes. Each array element is exactly 16-bits wide, even if the string contains only 7-bit ASCII characters. This widening of strings causes a memory usage increase. However, since almost all initial Allegro CL strings are stored in memory-mapped files, the initial runtime memory usage difference between International Allegro CL and non-international Allegro CL is less than 5%. Users wishing to deliver applications with their (read-only) strings in similarly memory mapped files can use the `:purify` option to **generate-application**. Please see *delivery.htm* for more information.

---

## 2.4 Character names

Lisp characters can be represented using the ``#[name]` syntax, where `[name]` is the character's Unicode name. Since the Unicode naming convention uses spaces in character names, and since the Lisp character reader treats space as a token delimiter, `#\_` (underscore) characters are used to act as spaces in the Unicode name. For example, the following shows the unicode name for `u+0141`:

```
> (code-char #x0141)
#\latin_capital_letter_l_with_stroke

> (format t "u+~4,'0x" (char-code #\latin_capital_letter_l_with_stroke))
u+0141
```

Not all Unicode characters have names. In particular, most CJK (Chinese/Japanese/Korean) characters are unnamed. If you are using Mule or the Allegro CL IDE to enter Japanese characters, though, you can name the characters directly:

```
> #\あ
#\hiragana_letter_a ; This character has a Unicode name.

> #\気
#\気 ; This character does not have a Unicode name.
```

Character names specified in ANSI Common Lisp are also recognized. Thus, some characters have more than one name in Allegro CL:

```
> (format t "u+~4,'0x" (char-code #\latin_capital_letter_a))
u+0041

> (format t "u+~4,'0x" (char-code #\A))
u+0041
```

---

---

## 3.0 External formats

As described above, International Allegro CL characters and character strings are represented internally using the Unicode standard with each character occupying exactly 16-bits per character code. Externally, however, and outside of Allegro CL's control, most non-ASCII characters are stored in variable-width multi-byte models using any one of several different representations.

For example, there are several common ways to represent Japanese characters, and most of these encodings specify that ASCII characters (which are non-Japanese) occupy a single 8-bit byte each, whereas Japanese characters may occupy two or three 8-bit bytes each depending on the character and the encoding.

Allegro CL provides stream-level and foreign-function call-level automatic translation between Unicode and several of these external formats. We describe how external-formats are used in this section as well as how users can define their own Unicode to External Format translations for their own external formats.

In the simple-stream implementation used by Allegro CL (see *streams.htm*), the basic unit is an *octet*, which is an 8-bit byte. Reading and writing functions work with octets. In this document, we typically describe behavior on octets.

---

## 3.1 External-Format Overview

### 3.1.1 Basic External-Format Types

The simplest external-format is for the Latin-1 character set. This is the external-format used when the default locale (sometimes known as the "C" or "POSIX" locale) is being used. For input, the Latin-1 external-format translation simply takes the next input octet and forms a Lisp character from the single octet's numeric value. For output, the external-format takes the Lisp character's code, and uses, as its octet output, the character code value. If the Lisp character code value is greater than 255 (i.e., what can be represented as a Latin-1 octet), then the ASCII value for question-mark (== 63) is used as the output octet. Thus question-marks appearing in Latin-1 output can indicate places where non-Latin-1 characters are used.

The next simplest class of external-formats are the 8-bit character sets, such as for any of the ISO-8859 sets. (The Latin-1 case described above is actually ISO-8859-1). For these external-formats, each Lisp character corresponds to one octet. (A special case exception is the `#\newline` case described below.) For the non-Latin-1 external-formats, the translation is generally done by fast table lookup. The following are the names and nicknames for the 8-bit external-formats supplied with Allegro CL:

Name	Nicknames	Comments
------	-----------	----------

```

-----
:latin1          :ascii, :8-bit, :iso8859-1, t
:1250            For MS Windows
:1251            For MS Windows
:1252            For MS Windows
:1253            For MS Windows
:1254            For MS Windows
:1255            For MS Windows
:1256            For MS Windows
:1257            For MS Windows
:1258            For MS Windows
:iso8859-2       :latin-2, :latin2
:iso8859-3       :latin-3, :latin3
:iso8859-4       :latin-4, :latin4
:iso8859-5       :latin-5, :latin5
:iso8859-6       :latin-6, :latin6
:iso8859-7       :latin-7, :latin7
:iso8859-8       :latin-8, :latin8
:iso8859-9       :latin-9, :latin9
:iso8859-14      :latin-14, :latin14
:iso8859-15      :latin-15, :latin15
:koi8-r

```

The following are special external-formats not listed above:

- `:emacs-mule`  
Used to communicate international characters through the Allegro CL Emacs-Lisp Interface.
- `:octets`  
Used simply to pass unconverted octets (8-bit bytes). Equivalent to `(crlf-base-ef :latin1)`. See [below](#) for a description of "base" external-formats and for an explanation about **crlf-base-ef**.
- `:void`  
This external-format signals an error (of type `void-external-format`). This external-format can be used, for example, as the **stream-external-format** of a stream meant only for binary I/O.

The general class of external-formats are for the variable-width multi-byte character sets often used for Asian languages. As described above, a single Lisp character may be represented externally using several external-format octets. The external-format conversion routines consume octets on input or generate octets on output, and may use table lookup for translation to/from Lisp characters. The following are the names and nicknames for the multi-byte external-formats supplied with Allegro CL:

Name	Nicknames	Comments
-----	-----	-----
<code>:utf8</code>	<code>:utf-8</code>	
<code>:big5</code>		
<code>:gb2312</code>		
<code>:euc</code>	<code>:ujis</code>	

:874	For MS Windows
:932	For MS Windows
:936	For MS Windows
:949	For MS Windows
:950	For MS Windows
:jis	
:shiftjis	

See [Section 3.1.2 The unicode and fat External-Format Types](#) for external formats that are exactly two bytes wide.

---

### 3.1.2 The unicode and fat External-Format Types

Two external forms that use precisely two bytes (16 bits) per character are `:fat` and `:unicode`. These external formats are similar, but `:unicode` follows unicode byte-ordering conventions. In particular, when a stream is first opened with the unicode external-format or when a stream's external-format is changed (via (**setf stream-external-format**)) to the unicode external-format, the unicode byte-order-marker is used in the following way:

The first time a character is requested from the stream (e.g., via `read-char`) a check for the unicode byte-order marker is made. If one is found, then the stream's internal state for subsequent character byte-ordering is set accordingly so that any necessary byte-swapping is done automatically by the external-format convertor. If a byte-order marker is not found, then little-endian ordering is assumed.

The first time a character is output to the stream (e.g., via `write-char`) a unicode byte-order marker is written before that first character. See **sniff-for-unicode**.

---

### 3.1.3 Composed External-Formats

The most abstract of external-formats in Allegro CL are the "composed" or "composing" or "wrapper" external-formats. (The terms "composed external-format" and "composing external-format" are used interchangeably in Allegro CL documentation.) Unlike the basic external-formats described above, which translate between Lisp characters and octets, a composing external-format provides translations between Lisp characters and (other) Lisp characters. Thus, a composed external-format is created by combining a *composer*, which is a character-to-character convertor with a *composee*, which is a character-to-octets convertor.

There are two types of composing external-formats in Allegro CL, macro-based and encapsulating-stream-based. The macro-based composing external-formats were introduced in Allegro CL 6.0 and are the type of composing external-formats created by the **compose-external-formats** macro. The function **find-composed-external-format**, introduced in Allegro CL, can be used to compose either macro-based or encapsulating-

stream-based composing external-formats.

Defining a macro-based composing external-format involves defining the convertor macros (described later in this document) using calls to the `composee` external-format's convertors. The macros are then combined using **`compose-external-formats`** (or to the function **`find-composed-external-format`**, which uses the **`compose-external-formats`** macro).

An encapsulating-stream-based composing external-format works on a stream by internally converting that stream into an encapsulating stream. The base stream of the encapsulation uses the macro-based `composee` external-format. The encapsulator stream operates by converting the characters received from or sent to the base stream.

The most widely used composed external-format, named `:e-crlf`, is an encapsulating-stream-based composing external-format, and is used to convert the Common Lisp `#\newline` character from/to the combination `#\return #\linefeed` combination. The `:e-crlf` external-format is used by default on the Windows platform where the textual convention is to end each line (regardless of character set encoding) with the ASCII octets 13 and 10 which represent 'carriage return' and 'linefeed' respectively.

On the Windows platform, the default external-format used (that which is specified by the current locale, see [Section 5.0 External formats and locales](#)) is a composition of the `:e-crlf` external-format and the locale-specific base external-format. For example, on US English Windows, the default base external-format is called `:1252-base` (the 1252 names the "code page" name used by Windows for US English -- this character set is effectively the same as Latin-1). In this Windows environment, Allegro CL creates and uses as default the composed external-format `(:e-crlf :1252-base)`. In other words, all Input/Output is filtered through a `#\newline` processor as well as base-level external-format. For Japanese Windows, where the default code page is 932 (corresponding to Japanese Shift-JIS), the base external-format is `:932-base`, and Allegro CL composes the `:e-crlf` external-format with the `:932-base` external-format to create `(:e-crlf :932-base)` as the Lisp's default external-format.

The `:e-cr` external-format converts `ascii `carriage return'` to/from `#\Newline`. Composing this external-format with a "crlf-base" external-format effectively turns this into an external-format for the Macintosh line-ending convention.

The function **`eol-convention`** allows determining and setting the end-of-line convention of a stream. The default `eol-convention` on the Macintosh is `:unix`. Thus Allegro CL does not automatically compose the ``cr'` external-format on Macintosh platforms as is done with the ``crlf'` external-format on Windows platforms. However, this is easily done by a programmer using **`eol-convention`**.

At Allegro CL startup time, when the global value of `*locale*`, is being set, the default external-format is set as the **`locale-external-format`** of `*locale*`. See also [Section 5.1 The initial locale when Allegro CL starts up](#).

The changes made to Allegro CL 6.0 (and kept in later releases) regarding `#\newline` handling can, in some cases, cause compatibility problems for code that was explicitly handling multi-character newline terminations. The special composing external-format `cr-crlf` is designed to work around these problems. See [Appendix B #\newline Discussion](#) below.

The function **crlf-base-ef** extracts the external format composed with the `:e-crlf` external-format or the `:crlf` external-format when passed such a composed external format.

## Windows/Unix Portability Notes

Most external-formats supplied by Allegro CL are set up to be used portably in either the Unix or Windows environments by acting as aliases to appropriate lower-level "base" external-formats. This effect can be seen by evaluating **find-external-format** as follows (we have left out the addresses and you may see other differences in the printed representations):

```
;; On Unix/Linux:
```

```
(find-external-format :jis)
==> #<external-format :jis-base>
```

```
;; On Windows:
```

```
(find-external-format :jis)
==> #<external-format :crlf-jis-base>
;; or
==> #<external-format [':e-crlf :jis-base]>
```

The **crlf-base-ef** function can be used to return an external-format which strips out any composed crlf processing. Thus, on either Unix or Windows, the following result occurs (we have left out some of the information printed):

```
(crlf-base-ef (find-external-format :jis))
==> #<external-format [(crlf-base-ef :jis)]>
```

Note that while one can specify the name of a "base" external-format directly to `find-external-format`, this use is not recommended since not only is Windows/Unix portability potentially sacrificed, but also external-format autoloading may not properly occur when specifying "base" external-formats. Example:

```
;; The following only works if the :jis external-format
;; is already loaded (or autoloading) into Lisp.
;;
(find-external-format :jis-base)
```

```
;; The following has the same effect as the preceding
;; call, but it is more portable, and will also perform
;; any necessary external-format autoloading.
;;
(crlf-base-ef (find-external-format :jis))
```

### 3.1.4 Defining External-Formats

An external-format object is defined in Lisp. Many external-formats are pre-defined for and distributed with Allegro CL. Occasionally, new external-formats may be made available after a release, either as patches or as supplemental lisp files.

Users can define their own external-formats using **def-external-format**. Except for encapsulating composing formats, a complete external-format object includes translation macros specified by **def-char-to-octets-macro** and **def-octets-to-char-macro**. These macros are used internally by Allegro CL to fill code templates that use external-formats. Pre-filled versions of these templates can be built and stored as auto-loaded fasl files using the function **generate-filled-ef-templates**.

---

### 3.1.5 Retrieving Existing External-Formats

The **find-external-format** function takes as its required argument a name and returns the external-format whose name, or one of whose nicknames, matches the argument. When the argument is `:default`, **find-external-format** returns the external-format associated with `*locale*` (the current locale, see [Section 5.0 External formats and locales](#)). If the external-format cannot immediately be found as defined in the Lisp, then an attempt is made to autoload the external-format definition. The string "ef-" is concatenated with the string name of the argument and passed to the Common Lisp **require** function. This effectively means that a module named ef-[name].fasl, where [name] is the argument to **find-external-format**, is sought and, if found, loaded. Using autoloading in this way allows Allegro CL to have in memory only those external-formats and translation tables that are needed.

The **find-composed-external-format** function takes two external-format arguments, a composer external-format (either macro-based or encapsulated-based), and a composee external-format and returns their composition.

If you are preparing an application for delivery to another computer (using **generate-application**), and the application will likely use arbitrary external formats, it is best if you ensure the external formats can be found. The easiest way to ensure this is to specify the *runtime-bundle* keyword argument (to **generate-application**) true (this produces a runtime bundle file to be shipped with the application). External formats can then be loaded from the runtime bundle file when they are needed. The only ones that will be present in the application image file are the ones loaded during the image build.

---

### 3.1.6 External-Format Runtime Mode

An external-format lacking translation macro definitions is said to be in *runtime-mode*. This means that the external-format exists (i.e., can be retrieved with **find-external-format**), and other aspects of the external-

format such as its nicknames can be retrieved, but unfilled code templates cannot be filled for that external-format. The reason one may want an external-format to be in runtime-mode is that if the code templates for an external-format have already been filled by, say, having previously used **generate-filled-ef-templates**, then the macro definitions and other structures needed by the macros at their expansion time can be deleted to save space. The **def-ef-switch-to-runtime** macro is used to name a function (or function object) that when funcalled clears structures used by the macros that are not needed when the external-format is in runtime mode. The Allegro CL **switch-ef-to-runtime** function switches an external-format to runtime mode.

When an external-format is autoloaded (see [Section 3.1.5 Retrieving Existing External-Formats](#) above), an attempt is also made to autoload the pre-filled external-format code templates. These pre-filled templates are stored in separate fasl files, usually with names that begin with 'efft-'. If the pre-filled templates are successfully loaded, then the just-loaded external-format is automatically switched to runtime mode.

## 3.2 External-Format Usage

### 3.2.1 Streams

The Lisp 'open' function (and, analogously, 'load' and 'compile-file') takes an 'external-format' argument. If this argument is not specified, a default external-format, based on the current locale (see [Section 5.0 External formats and locales](#) below) is used.

When an operation requests to read an input character stream's next character, the stream's external-format template(s) will request one or more octets from the buffered stream device which it then translates into a Lisp character. Similarly, for writing Lisp characters via streams, the external-format is used to translate the Lisp character code in Unicode to the octet(s) specified by the external-format translation.

A stream's external-format can be changed at arbitrary times, using `(setf (stream-external-format ...) ...)`. If it is changed to be an external-format for which readers/writers are not already built, the Lisp compiler is invoked to build a new associated reader/writer in the stream for that external-format. Since the external-format translation routines are defined using macros, the Lisp compiler is used to build new readers/writers, thus keeping runtime stream overhead from external-format processing to the bare minimum.

### 3.2.2 String <-> External-Format Lisp Arrays

When defining a new external-format, the **string-to-octets** and **octets-to-string** functions are a convenient way to test the conversion macros. These functions (formerly known in Allegro CL 5.0.1 as **string-to-mb** and **mb-to-string** which are still supported aliases) and the functions **string-to-native** and **native-to-string** take an external-format argument to specify how to convert between a Lisp string and a Lisp octet array.

For example, the following translates to the Shift-JIS external-format:

```
> (setq mb (string-to-octets (coerce '(#\hiragana_letter_a
                                     #\hiragana_letter_i
                                     #\hiragana_letter_u)
                                     'string)
                               :external-format :shiftjis))

#(130 160 130 162 130 164 0)
7
```

The following takes the above Shift-JIS result and converts it to EUC:

```
> (string-to-octets
   (octets-to-string mb :external-format :shiftjis)
   :external-format :euc)

#(164 162 164 164 164 166 0)
7
```

The first value returned by `string-to-octets` is the octet array in EUC format. The second value is the number of octets generated including the null-terminating 0 which is added by `string-to-octets` (and not by the `external-format`).

### 3.3 External-Formats in 8-bit Lisp.

External-Formats are defined to convert octets to/from Lisp Unicode characters. Thus, they are only fully supported in the standard Allegro CL International (i.e., 16-bit) images, which are those using the 16-bit executable like *mlisp* and *alisp*. (On Windows, the executables have extension *exe*.) See *Allegro CL Executables* in *startup.htm* for information on the various images and executables.

But external-formats are also available, in a limited way, in non-international (i.e., 8-bit) Lisps (those using the 8-bit executable like *mlisp8* and *alisp8*).

Their usage is limited because it is an error in an 8-bit Lisp to create a Lisp character with a code exceeding 255. For example, one cannot use an external-format to create a Unicode Japanese Lisp character in an 8-bit Lisp. One can safely, however, use the latin-1 external-format in an 8-bit Lisp (which is, in fact, the default external-format in an 8-bit Lisp) without any problems because it never creates a character with code exceeding 255. Multi-Byte external-formats, such as utf-8, can also be used, but only for characters whose **char-code** does not exceed 255. Furthermore, the `:crlf` composing external-format can also be used (and is, in fact, used on Windows platforms) because it only deals with ASCII character codes.

As described in [the earlier section on Unicode](#) in this document, a non-Latin1 8-bit character is represented in an 8-bit Lisp with its character code being its native 8-bit (non-Unicode) encoding. In other words, no conversion between octets and Lisp character char-codes is meant to take place. This is, in fact, exactly how the Allegro CL latin1 external-format operates. The latin1 external-format is in effect a pass-through convertor. Thus, even if an 8-bit Lisp application knowingly handles non-Latin1 Lisp characters, there is generally no need to specify an external-format other than the default, latin1, external-format.

---

### 3.4 Older Allegro CL External-Format Compatibility

Use of the existing special `*default-external-format*` is discouraged in Allegro CL. Users are encouraged either to bind a locale to `*locale*` (see [Section 5.0 External formats and locales](#)) or to directly specify the desired external-formats when calling functions that take the `:external-format` argument (e.g., **with-native-string**, **string-to-octets**, **octets-to-string**, etc.) In Allegro CL, the default value of `*default-external-format*` is `:default`. When **find-external-format** is invoked with `:default`, the returned external-format will be that stored in `*locale*`.

---

## 4.0 Foreign-Functions

In C, the 'char' type is equivalent to an 8-bit byte. A C string is represented as an array of 8-bit bytes terminated by the null (or 0) byte. Therefore, a C routine expecting a 'char \*' argument may expect a null-terminated 8-bit character array (i.e., string) in this format.

The Allegro CL Foreign-Function Interface allows users/programmers to pass Lisp strings to C routines expecting 'char \*' arguments. Since Allegro CL internally null-terminates lisp string objects, passing a lisp string to a foreign function simply means internally passing the address to the first character of the lisp string's array.

The previous Allegro CL string-passing mechanism described above breaks down in the International version since the internal character codes of the lisp string's array are in Unicode, and non-ASCII characters may not match the codes in the locale's native (or external) format. Furthermore, even for ASCII-only strings, a 'char \*' argument expects its value to be in a format where ASCII characters are 1-byte per character. International Allegro CL represents all characters, including ASCII characters, as 2-bytes per character. The upper byte of each ASCII character is always zero. Therefore, even if a user wishes to pass an ASCII-only string from Allegro to a foreign function, the foreign function will most likely treat the string argument as truncated since the first upper (all-zero) byte will be regarded as the string terminator.

One solution offered to this problem was to provide a macro, called **with-native-string**, to be used around all foreign-function calls that pass strings. This macro is used to convert string arguments to native format using a dynamic-extent array of 8-bit bytes.

Even with the **with-native-string** solution, users porting foreign function code from earlier releases of Allegro CL to International Allegro CL would have to manually hunt down every string-passing foreign function call in order to wrap those calls with **with-native-string**.

In order to save users from this burden, Allegro CL has a keyword argument `:strings-convert` to **def-foreign-call**. The default value of this argument is `true`.

**def-foreign-call** creates a low-level function that actually calls out to the foreign code. When `:strings-convert` is `true`, arguments declared directly or indirectly as `(* :char)` at `def-foreign-call` time are handled specially. The low-level function is augmented so that for each `(* :char)` declared argument, a check is made at runtime to see if that declaration's corresponding value is a string. If it is, then that value is converted at runtime to native-string format using a dynamic-extent array, and this new array is passed in place of the original string argument to the foreign function call.

Since this runtime search for string arguments only happens for those arguments declared directly or indirectly as `(* :char)`, no new code is introduced for foreign functions not expecting strings. Consequently, no checking is introduced if the arguments are specified as a `&rest` list.

Suppose we have a C function which takes two input string arguments and one output string argument defined as follows:

```
/*
 * concatenates first two arguments,
 * returns result in third output argument.
 */
char *
myconcat(char *st1, char *st2, char *retval)
{
    int st1len = strlen(st1);
    int st2len = strlen(st2);
    int i;

    for (i = 0; i < st1len; i++) {
        retval[i] = st1[i];
    }
    for (i = 0; i < st2len; i++) {
        retval[st1len + i] = st2[i];
    }
    retval[st1len + st2len] = '\0';
    return retval;
}
```

To call this function from Allegro CL using the foreign function interface, one can define the Lisp foreign function as follows:

```
(ff:def-foreign-call myconcat ((st1 (* :char)))
```

```

      (st2 (* :char))
      (result (* :char)
              (vector (unsigned-byte 8))))
:returning :int)

```

Evaluating the above form causes the following warnings (of condition type `ff:strings-convert-def-warning`) to be signaled:

Warning: A runtime `with-native-string` call is being generated for argument ``st1'` to the foreign-function ``myconcat'`. The `with-native-string` macro can be used for explicit string conversions around the foreign calls. This warning is suppressed when `:strings-convert` is specified as `nil` in the `def-foreign-call`.

Warning: A runtime `with-native-string` call is being generated for argument ``st2'` to the foreign-function ``myconcat'`. The `with-native-string` macro can be used for explicit string conversions around the foreign calls. This warning is suppressed when `:strings-convert` is specified as `nil` in the `def-foreign-call`.

Warning: While defining ``myconcat'`: Automatic string conversion suppressed for argument ``(result (* :char) (vector (unsigned-byte 8)))'` since a `lisp` type is specified. The `with-native-string` macro can be used for explicit string conversions around the foreign calls. This warning is suppressed when `:strings-convert` is specified as `nil` in the `def-foreign-call`.

Disregarding the warnings for the moment, and continuing on to use the just-defined foreign function, note that we can call it with `lisp` string arguments:

```

user(22): (let ((x (make-array 500 :element-type '(unsigned-byte 8))))
           (myconcat "abc" "def" x)
           (octets-to-string x))
"abcdef"
6
6

```

The returned value, `"abcdef"`, which is the concatenation of the first two arguments (performed by the C foreign function) is correctly returned.

To turn the example above into one which (a) doesn't generate the warnings, and (b) generates faster runtime

code since string conversion checking will be suppressed, one can set the *strings-convert* keyword argument to false as follows:

```
(ff:def-foreign-call myconcat ((st1 (* :char))
                               (st2 (* :char))
                               (result (* :void)))
  :strings-convert nil
  :returning :int)
```

By specifying `:strings-convert` to `nil`, the foreign-function interface will not automatically convert string arguments. Thus, to call the foreign-function defined this way, one needs to pass converted string arguments as follows:

```
user(23): (let ((x (make-array 500 :element-type '(unsigned-byte 8))))
  (with-native-string (st1 "abc")
    (with-native-string (st2 "xyz")
      (myconcat st1 st2 x)))
  (octets-to-string x))
"abcxyz"
6
6
```

It is instructive to note what happens when `:strings-convert` is `nil`, yet the string arguments are not converted:

```
user(24): (let ((x (make-array 500 :element-type '(unsigned-byte 8))))
  (myconcat "abc" "def" x)
  (octets-to-string x))
"ad"
2
2
```

The result is the first character of the first string concatenated with the first character of the second string. The reason this happens is that the C foreign function sees the unconverted arguments as Unicode strings with each element being two octets wide. To the C function, each argument appears as the first octet being an ASCII character, and the second octet being a string NULL terminator.

Note that on big-endian platforms, the result of the above form is "" (i.e., the empty string). That's because the Unicode Ascii values have zero in their upper-bytes, and any array of Unicode Ascii values appear to C routines as being zero-length null-terminated strings.

Note that neither setting of the *strings-convert* keyword argument affects foreign function return results or "output" variables. Users with foreign function code that expects to "fill in" Lisp strings directly will need to modify those calls to pass octet arrays and do conversions, e.g., with **octets-to-string** as shown above for the example's third argument.

## 5.0 External formats and locales

As described in the section [Section 5.1 The initial locale when Allegro CL starts up](#), the global variable `*locale*` is bound to a locale object. The value of `*locale*` is treated as the current locale.

The Windows and UNIX Operating Systems define a locale environment for each running program. The OS definition of locale describes date/time formats, currency printing formats, and sort ordering information in addition to character type information.

`*locale*` is used to determine the default external-format (see example just below). The external-format used in the default Lisp locale object is derived from the encoding.

Here is an example showing how changing the locale changes the external-format.:

```
cl-user(47): (dolist (x (list (find-locale "C")
                             (find-locale "japan.EUC")))
              (let ((*locale* x))
                (format t "~&*locale*~s;~%default external-format=~s~2%"
                        *locale*
                        (find-external-format :default))))
*locale*=#<locale "C" (English/default) [:latin1-base] @ #x[...]>;
default external-format=#<external-format :latin1-base @ #x[...]>

*locale*=#<locale "japan" [:euc-base] @ #x[...]>;
default external-format=#<external-format :euc-base @ #x[...]>
```

See [Section 5.1 The initial locale when Allegro CL starts up](#) for information on how the initial locale (that is, the initial value of `*locale*`) is set.

The `*locale*` variable is analogous to the Common Lisp `*package*` variable in that rebinding the variable can affect basic Lisp functionality such as Input/Output.

The standardized convention for locale names is `Name[_Territory][.Charset]`.

Suppose the following Lisp session were started in a Japanese locale using the EUC encoding. One can override the default external-format by dynamically changing the locale as follows:

```

> *locale*
#<locale "japan" [:euc-base] @ #x1001dc7522>

> (string-to-octets "日本語")
#(198 252 203 220 184 236 0)      ;;; <<< This is the EUC encoding.
7

> (let ((*locale* (find-locale "japan.shiftjis")))
      (format t "(find-external-format :default) = ~s~%"
                (find-external-format :default)
                (string-to-octets "日本語")))
(find-external-format :default) = #<external-format :shiftjis-base ...>
#(147 250 150 123 140 234 0)      ;;;; <<<< This is the Shift-JIS encoding.
7

```

## 5.1 The initial locale when Allegro CL starts up

The initial locale (that is, the initial value of `*locale*`) can be determined in various ways. Note that the variable is first set to a locale that (presumably) always exists. It is only reset if valid locales are determined from the additional steps.

- Initially, `*locale*` is set to `(find-locale "C")` (see **find-locale**). This becomes the default value for `*locale*` if the following tests fail.
- If the environment variable `ACL_LOCALE` is set, then Allegro CL attempts to look up, using **find-locale**, the Lisp locale object named by `ACL_LOCALE`. If a corresponding lisp locale object is found, then `*locale*` is set to this object.
- If the environment variable `ACL_LOCALE` is not set, then Allegro CL attempts to look up, using **find-locale**, the Lisp locale object named by a call to `setlocale(LC_CTYPE)`. (This is the portable Operating Systems level way to look up a locale on both Windows and Unix.) If a corresponding lisp locale object is found, then `*locale*` is set to this object. **Note:** the Operating System is polled, as described in this step, only if `ACL_LOCALE` is not set. If `ACL_LOCALE` is set but its value is bogus (i.e. **find-locale** returns `nil` on the value) the value of `*locale*` is its initial value `(find-locale "C")` and will only be changed, if at all, by the next step.
- If the `-locale` command-line argument is specified (see *Command line arguments* in *startup.htm*), then Allegro CL attempts to look up, using **find-locale**, the lisp locale object named by the argument value. If a corresponding lisp locale object is found, then `*locale*` is set to this object. Thus, using `-locale` effectively overrides any environmental setting of `LC_CTYPE` or `ACL_LOCALE`. Note that this step is performed when command-line argument processing is done. All the steps above are done earlier in the startup procedure. See *What Lisp does when it starts up* in *startup.htm* for details.

### Some examples from UNIX

```
% env ACL_LOCALE=japan.EUC mlisp
```

```

cl-user(1): *locale*
#<locale "japan" [:euc-base] @ #x404a02da>

% mlisp -locale cs_cz
cl-user(1): *locale*
#<locale "cs_CZ" [:iso8859-2-base] @ #x400f7a02>

% env ACL_LOCALE=japan.EUC ./lisp -I mlisp.dxl -locale cs_cz
cl-user(1): *locale*
#<locale "cs_CZ" [:iso8859-2-base] @ #x400f7a02>
cl-user(2):

```

On UNIX machines, you can determine the available locales with the **locale -a** shell command. A process' locale is often specified by setting the LANG environment variable, which often automatically sets several other variables including one named LC\_CTYPE.

---

## 5.2 Locales in applications

If you are preparing an application for delivery to another computer (using **generate-application**), and the locale on the computer being delivered to is different from the locale on the computer generating the application, you must be sure the application can successfully change locales. The easiest way to ensure this is to specify the *runtime-bundle* keyword argument (to **generate-application**) true (this produces a runtime bundle file to be shipped with the application from which the locale-changing code can be loaded if needed).

If you know the specific locale on the target computer, you can call **find-locale** with that locale name (a string) as the argument during the image build (put such a form in a file and include that file as one of the list of files which is the value of the *lisp-files* argument to **build-lisp-image** or the *input-files* required argument to **generate-application**). But if you want to be ready for any locale, specify *runtime-bundle* true.

---

## 6.0 Localization support in Allegro CL

*Localization* is the process of modifying a program or application so that attributes specific to the location where the program or application is being run are used. Such attributes include such things as how dates are represented (day/month/year vs month/day/year) and the glyph for currency (e.g. \$ meaning United States dollars) and whether the symbol appears before or after the amount.

This section and its subsections describe localization support in Allegro CL.

---

## 6.1 Introduction to locales

A locale is a Lisp object which contains linguistic, cultural, and governmental rules and conventions. The Lisp locale concept is based on the C-based POSIX locale specification. At runtime, Lisp locales operate independently of any active POSIX locales except that at Lisp startup time, the initial default Lisp locale (stored in the variable `*locale*`) is set using the process' current POSIX locale setting.

A POSIX locale has attributes from several categories. The following list shows the major categories, including a brief overview description of how they are used by Allegro CL. Further details follow the list.

- `LC_MONETARY`: Used to specify currency display information. Allegro CL supports this by defining a function, **locale-print-monetary**. A related function, **locale-format-monetary**, can be used with Lisp format's `~/` directive. The principal argument to these functions (and the directive) is a number. See **locale-format-monetary** description for more information.

There are no plans at this time to support parsing monetary information.

- `LC_NUMERIC`: Used to specify locale-dependent numeric display information (i.e., thousands separator, grouping, and decimal point indicator). Allegro CL supports this by defining a function, **locale-print-number**. A related function, **locale-format-number**, can be used with Lisp format's `~/` directive. The principal argument to these functions (and the directive) is a number.

The Allegro CL Lisp reader will not be modified to parse numbers in locale numeric display format, but a new function, **locale-parse-number**, is added to create a Lisp number from a string representing a number in the locale numeric display format.

- `LC_TIME`: Used to specify locale-specific date/time display information. Allegro CL supports this by defining the function **locale-print-time**. A related function, **locale-format-time**, can be used with Lisp format's `~/` directive. The principal argument to these functions (and the directive) is a universal time (as returned by **get-universal-time**).
- `LC_COLLATE`: Used to specify character collation sequences. At this time, Allegro CL does not support this category. Allegro CL collation uses Unicode Collation Element tables which are specified independently from locales. (See [Section 7.0 String collation with international characters](#) for more information).
- `LC_CTYPE`: Used to specify charset information for non-ASCII characters. This category is not used in Allegro CL where all characters are represented in Unicode.
- `LC_MESSAGES`: Used to specify translation texts. At this time, Allegro CL does not have plans to support this category.

### Other Categories

A locale may include other categories (e.g., for specifying postal address information, salutations, etc.). Allegro CL does not provide any particular support for such other categories except to retain their information in the locale object and make it available to user programs via **locale-attribute**.

The names and meanings of locale attributes are not given in this document. They are available from Operating System Documentation. On LINUX, `man 5 locale`. On other UNIX, `man 4 locale`, or `man localedef` provide the information. On Windows, the `localeconv` in the Run-Time Library Reference in the MSDN documentation has it.

---

## 6.2 Locale Definition

In Allegro CL, a locale is effectively a set of attribute/value pairs. These pairs are sectioned into categories as defined by POSIX. Each attribute name is unique in the locale. In other words, within a single locale, no category can specify an attribute with the same name as a different category's attribute (this requirement seems to be specified by POSIX). In addition to attribute/value pairs, a Lisp locale also holds slots containing (1) the locale's name, and (2) the locale's default external-format.

Lisp locales are defined in Lisp using **load-localedef**. The **load-localedef** function takes two arguments: A pathname and a localename. The default for the localename is the pathname's name field. The pathname names a localedef source file. Several localedef source files are included with Allegro CL. These definitions come from the IBM Universal Locales project. They can be found in the directory specified by `*locales-dir*`.

We do not document the format of a localedef file, although platform operating system documentation may be available. On Linux platforms, one can look at the man entries for `locale(5)` and `localedef(1)`. The localedef file format is in ASCII, and is mostly self-descriptive in case a user wishes to make local customizations.

The localedef file does not include external-format information. After **load-localedef** parses the input file and creates the appropriate attribute/value pairings, the external-format is determined by looking to see if a charset is specified in the locale name (using the `".[charset]"` specification). If the name does not specify the external-format, then Allegro CL uses a `[language -> external-format]` table which is kept in Lisp.

Locales are found with **find-locale**. The function **find-locale** takes a single string-designator argument, *name*, and returns the locale specified by *name*. Please see the documentation page of **find-locale** for information about how locales are returned and/or created based on its argument.

A locale is a class. Accessors to slots include **locale-name** and **locale-external-format**.

The function **locale-attribute** can be used to access an attribute of a locale.

The function **merge-locale-categories** enables users to create a locale object which combines all of a specified locale with categories from other existing locales. Such a locale is unnamed (i.e., its name slot is nil), and is never returned by **find-locale**.

---

## 6.3 Locale Attribute Accessors

A locale object holds the attributes conceptually in the same way as specified in the localedef file. As specified above, the locale-attribute function can be used to obtain the value of any attribute. However, since some of the attribute names and values are oriented to C programming, Allegro CL provides a more Lisp-appropriate interface to the attributes used in the localedef files. This alternate interface provides the following two features:

- For attribute names, the dash character is used instead of the underscore character. For example,

```
(locale-int-curr-symbol locale)
== (locale-attribute locale "int_curr_symbol" :category
"LC_MONETARY" )
```

- Accessors for known boolean attributes, which are specified in the localedef file using 0 or 1, return Lisp true and false. For example,

```
(locale-p-cs-precedes locale)
== (eql 1 (locale-attribute locale "p_cs_precedes" :category
"LC_MONETARY" ) )
```

Note that attributes with multiple-values, such as for the list of abbreviated months, are specified in the localedef file using the semi-colon (;) delimiter. When parsed by **load-localedef**, these values create Lisp lists. Also, string characters are represented using Unicode. For example, the ``abmon'` (abbreviated month names) category may be specified in a localedef as follows (`'` is the escape character):

```
abmon      "<U004A><U0061><U006E>" ; "<U0046><U0065><U0062>" ; /
           "<U004D><U0061><U0072>" ; "<U0041><U0070><U0072>" ; /
           "<U004D><U0061><U0079>" ; "<U004A><U0075><U006E>" ; /
           "<U004A><U0075><U006C>" ; "<U0041><U0075><U0067>" ; /
           "<U0053><U0065><U0070>" ; "<U004F><U0063><U0074>" ; /
           "<U004E><U006F><U0076>" ; "<U0044><U0065><U0063>"
```

The lisp value for the abmon slot in a corresponding Lisp locale would be as follows:

```
(locale-attribute locale "abmon" :category "LC_TIME" )
```

```
==>
```

```
("Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec" )
```

Note that, in particular, POSIX locales define `LC_NUMERIC `grouping'` and `LC_MONETARY `mon_grouping'` as strings. In Lisp, these categories are defined as lists, or, if the category has a single value, an integer. A -1 value in these grouping attributes corresponds to the POSIX `CHAR_MAX` specification.

These are pre-defined user-visible accessor definitions:

- **locale-int-curr-symbol**

- **locale-currency-symbol**
  - **locale-mon-decimal-point**
  - **locale-mon-thousands-sep**
  - **locale-mon-grouping**
  - **locale-positive-sign**
  - **locale-negative-sign**
  - **locale-int-frac-digits**
  - **locale-frac-digits**
  - **locale-p-cs-precedes**
  - **locale-p-sep-by-space**
  - **locale-n-cs-precedes**
  - **locale-n-sep-by-space**
  - **locale-p-sign-posn**
  - **locale-n-sign-posn**
  - **locale-am-pm**
  - **locale-mon**
  - **locale-abmon**
  - **locale-day**
  - **locale-abday**
  - **locale-t-fmt-ampm**
  - **locale-t-fmt**
  - **locale-d-fmt**
  - **locale-d-t-fmt**
  - **locale-grouping**
  - **locale-thousands-sep**
  - **locale-decimal-point**
- 
- 

## 7.0 String collation with international characters

Allegro CL provides string comparison functions such as **string<** and **string>**. These functions operate by doing binary comparisons of their argument arrays using the **char-code** values of the characters in the arrays. The effect of these functions is to compute the lexicographic order of their argument strings based on the array element char-code (Unicode) values.

For English-only characters, Unicode order is alphabetic except that upper case characters have lower code value than lower case characters. Thus, one problem with simple **string<** comparisons, which may concern some users, is that any string beginning with an uppercase letter will always be ordered before any string beginning with a lowercase letter.

Another problem is that with accented, or unicode-combined characters, simple **string<** ordering causes all such characters to be ordered after all Ascii characters.

Further problems include that different cultures order the same letters differently; that in some languages characters may be combined to form a single letter (e.g., "ch" in Spanish); and other characters may be equivalent to two or more letters (e.g., the "ae" ligature).

To address these problems, Allegro CL supports tools which parse and utilize Unicode Collation Element Tables for string collation. The Unicode Organization has defined a default ordering. Allegro CL uses this ordering by default.

The collation ordering can be customized by creating and loading alternate unicode collation element tables into Lisp. (The current Default Unicode Collation Element Table is provided in the following data file: <http://www.unicode.org/unicode/reports/tr10/allkeys.txt>.) The tables themselves are Ascii files. Their format is not documented here, but is described in the Unicode Technical Standard #10. Links to versions of that document are given in [Appendix D Links to Unicode Reports](#). The **parse-ucet** function can be used to load one or more collation element tables into Lisp (which will be in addition to the one already present for default Unicode collation behavior). The **parse-ucet** function creates a Lisp ucet (Unicode Collation Element Table) object.

The function **string-sort-key** takes as arguments a string and a ucet object (as well as additional arguments). The function returns a string. The main property of **string-sort-key**'s return value is that it can be used as an ordering key for **string<** or **string>**. In other words, if string A is to be collated before string B relative to the table which is the value of `ucet`, then

```
(string< (string-sort-key A :ucet ucet) (string-sort-key B :ucet ucet))
```

is true.

If the `:ucet` argument is not specified, `(string-sort-key A)` returns a sort key relative to the Default Unicode Collation Element Table, already present in Lisp. Because this default Lisp ucet object already exists for Allegro CL, and there is thus no need to use **parse-ucet** for default Unicode collation behavior; it is only needed when you wish to define your own collation.

## 8.0 Earlier International Allegro CL Compatibility

### 8.1 EUC Module

In previous releases of International Allegro CL for UNIX, EUC was the only supported external-format, and a special internal format known as 'process-code' was used. Some user-visible euc-specific functionality was added to the Lisp when International Allegro CL was first created in Release 6.2 on UNIX. This functionality, which mostly consists of character type definitions listed below, is moved into a special module no longer built into Allegro CL. This way, backward compatibility can be achieved by loading this module using `(require :`

auc).

The following symbols name type specifiers which correspond to their named EUC codesets in the deprecated functionality for the EUC external-format. These remain defined for backward compatibility.

- `ascii`
  - `codeset-0`
  - `codeset-1`
  - `codeset-2`
  - `codeset-3`
  - `gaiji`
  - `half-size-kana`
  - `half-size-katakana`
  - `half-sized-kana`
  - `half-sized-katakana`
  - `kanji`
- 

## 8.2 `:mode` Option Removal

Because of incompatibilities between the UNIX and Windows Operating Systems with respect to textual line termination, a special keyword argument, `:mode` was added to the **`cl:open`** function. This flag determined whether a stream would read one or two characters when a text line was terminated. With `#\newline` handling integrated into external-formats, this flag is no longer needed. A warning is signaled if this flag is used. See also [Appendix B `#\newline` Discussion](#).

---

## Appendix A: Functions, Symbols, Variables Documentation

Individual operators, variables, etc. are documented on their own pages, as is standard in Allegro CL documentation. In this section, we provide a list of the operators, variables, etc. with brief descriptions and links to the documentation pages.

---

### Appendix A.1 External-Format API

- **`def-external-format`**, a macro which defines an external-format object. External-formats are structs (defined using **`defstruct`**).
- **`def-ef-switch-to-runtime`**, a macro which associates a function object with an external-format.
- **`switch-ef-to-runtime`**, a function which invokes the function object **`def-ef-switch-to-runtime`** for an

external format.

- **find-external-format**, a function which returns the external-format object named by the argument symbol.
- **find-composed-external-format**, a function which returns the composed external-format object (either macro-based or encapsulating-streams-based) named by the argument external-formats.
- **def-char-to-octets-macro**, a defining macro which defines a macro associated with an external format which will be used for converting a character object to a sequence of octets.
- **char-to-octets**, a macro which expands to the macro stored in the `char-to-octets-macro` slot of the external-format passed in.
- **def-octets-to-char-macro**, a defining macro which defines a macro associated with an external format which will be used for converting a sequence of octets to a character object.
- **octets-to-char**, a macro which expands to the macro stored in the `octets-to-char-macro` slot of the external-format passed as an argument.
- **compose-external-formats**, a macro which creates a new external-format composed of argument external-formats.
- **composed-external-format-p**, a function which returns true or false as its argument is or is not a composed external format.
- **ef-composer-ef**, a function which returns the value in the `composer` slot of an external format.
- **ef-compose-ef**, a function which returns the value in the `compose` slot of an external format.

The following two functions are named by unexported symbols. We document them because the sources for Allegro CL external-formats will be made available to users and these functions and trie data structures are referenced in the sources. The symbols naming these functions are kept internal in the Allegro CL packages to indicate that their associated functions are subject to change.

## build-trie

### Function

**Package:** `excl`

**Arguments:** `&key name list index-key value-key optimize`

*name* should be a symbol. *list* should be a list of index/value pairs. *index-key*, and *value-key* should be designators of functions of one argument (symbols or function specs, or function objects). *optimize* should be a boolean.

**The symbol naming this function is not exported.** `excl::build-trie` builds a trie data structure consisting of the data from *list*. A trie data structure holds index/value pairs where most of the values are the same default value. The trie structure does not store these default values for each index, thus saving space and making lookup more efficient. It is not necessary for users to know details of trie data structures.

The *list* argument names a list of index/value pairs for the trie. The *index-key* is a function which when applied

to a pair returns the index of the pair. The value-key is a function which when applied to a pair returns the value of the pair.

If the optimize argument is true, then any rows in the resulting trie that would be equalp to any rows in any of the tries returned by excl::all-tries are shared. The result is that all equalp rows of all existing tries which are equalp become eq.

## Examples:

```
[*package* is the excl package for these examples]
```

```
(let ((jis-to-unicode-list '((#x2121 . #x3000)
                             [...]
                             (#x2124 . #xff0c)
                             [...])))
      (build-trie :name :jis-to-unicode
                  :list jis-to-unicode-list
                  :index-key #'car
                  :value-key #'cdr)
      (build-trie :name :unicode-to-jis
                  :list jis-to-unicode-list
                  :index-key #'cdr
                  :value-key #'car))

(let ((unicode-to-jis-trie
      (cadr (member :unicode-to-jis (excl::all-tries)))))
      (write (aref (aref unicode-to-jis-trie
                       (ldb (byte 8 8) #x3000))
                (ldb (byte 8 0) #x3000))
            :base 16))
      ==> [prints 2121])
```

---



---

## all-tries

### Function

**Package:** excl

**Arguments:**

**The symbol naming this function is not exported.** `excl::all-tries` returns a list of all tries built by `excl::build-trie`. The returned list is in plist format: `(trie-name1 trie1 trie-name2 trie2 ...)`.

---



---



---

## Appendix B: #\newline Discussion

ANSI Common Lisp specifies that the single character #\newline denotes the end of a character text line. This requirement is complicated by the lack of a uniform convention among Operating Systems regarding textual line endings. To end a line, UNIX based applications generally uses Ascii 10; Macintosh based, Ascii 13; and Windows based applications use two Ascii characters: 13 followed by 10.

To confuse things further, Common Lisp implementations also differ on which Ascii character code is used for #\newline. Some use 13, others use 10. Some Common Lisp implementations may have disregarded the ANSI requirement that a single newline character be returned at the ends of lines, and for Windows, where two character bytes denote line endings, return more than one character at the end of a line.

Allegro CL 5.0.1 uses Ascii 10 for #\newline, following the UNIX convention. By using UNIX compatibility modes provided by MS Windows, Allegro CL 5.0.1 is able (via the ':mode :text' option to the Allegro CL 'open' function) to read/write files following the Windows line ending conventions. One problematic side-effect with this approach, though, is that the Allegro CL **file-position** routine is inaccurate.

Allegro CL has an external-format processing mechanism for handling character I/O. Using this mechanism, developed initially for multi-byte international characters, several adjacent external bytes may translate to a single Common Lisp character. A natural use of this mechanism is to map external end-of-line markers to/from the single Common Lisp #\newline character. This mechanism eliminates the need for the ':mode :text' option to the **open** function and also fixes the **file-position** problem.

Specifically, for Windows the newline processing is handled using a composing external-format. The exact description is as follows:

Standard Translation:

-----

Characters -> External Octets:

Lisp Character	External Octet Sequence
-----	-----
#\return	-----> 13
#\newline [eq #\linefeed]	-----> 13 10

External Octets -> Characters:

External Octet Sequence	Lisp Character
-----	-----

```

13 10 -----> #\newline [eq #\linefeed]
13 -----> #\return
10 -----> #\linefeed [eq #\newline]

```

One effect of this translation is that since `#\linefeed` is the same as `#\newline`, `'#\return #\linefeed'` translates to `'13 13 10'` instead of `'13 10'` as would happen with Allegro CL 5.0.1 on Windows. This scenario is only likely to be noticed if a program deliberately inserts the `'#\return #\linefeed'` sequence into a string that is to be converted to external format.

A fix for this situation is for the user/programmer simply to use `'#\newline'` instead of the `'#\return #\linefeed'` sequence. For users/programmers not immediately able to make this change, a compatibility mode exists in the form of an alternate composing external-format which translates as follows:

Compatibility Translation:

Characters -> External Octets:

Lisp Character Sequence	External Octet Sequence
-----	-----
[ADD] #\return #\linefeed [eq #\newline]	-----> 13 10
#\return	-----> 13
#\newline [eq #\linefeed]	-----> 13 10

External Octets -> Characters:

External Octet Sequence	Lisp Character Sequence
-----	-----
[ADD] 13 13 10	-----> #\return #\return #\linefeed
13 10	-----> #\newline [eq #\linefeed]
13	-----> #\return
10	-----> #\linefeed [eq #\newline]

Note that the new rules allow the external octet sequence `'13 13 10'` to be preserved after undergoing a round-trip conversion via Lisp characters.

Use of the compatibility mode is only supported at lisp startup time via a new `-compat-crlf` command-line argument (see *Command line arguments* in *startup.htm*).

## Appendix C: 8-bit images

Standard Allegro CL supports international characters and uses two bytes (16 bits total) for each character. 8-bit characters are not supported in standard Allegro CL. However, 8-bit versions of Allegro CL are supplied. The 8-bit executables have `8' in their names (*mlisp8*, etc.) While most users will likely use the standard version, some (particularly those who manipulate very large ASCII strings) may wish to use the 8-bit version. Note that the 8-bit version does not support 16-bit strings or characters and fast files are incompatible between the two versions.

---

---

## Appendix D: Links to Unicode Reports

The Unicode technical reports vary slightly with each new release of the Unicode Standard. At the time of this writing, the variations have not mattered to Allegro CL, but to avoid possible confusion, we provide two links for UTS #10: Unicode Collation Algorithm (hereafter, TR 10):

- The version of TR 10 used for this Allegro CL release: <http://www.unicode.org/unicode/reports/tr10/tr10-6.html>.
- The current version of TR 10 (currently not significantly different from the above version): <http://www.unicode.org/unicode/reports/tr10/>.