NAME

Unicode::UCD - Unicode character database

SYNOPSIS

```
use Unicode::UCD 'charinfo';
my $charinfo   = charinfo($codepoint);

use Unicode::UCD 'casefold';
my $casefold = casefold(0xFB00);

use Unicode::UCD 'casespec';
my $casespec = casespec(0xFB00);

use Unicode::UCD 'charblock';
my $charblock  = charblock($codepoint);

use Unicode::UCD 'charscript';
my $charscript = charscript($codepoint);

use Unicode::UCD 'charblocks';
my $charblocks = charblocks();

use Unicode::UCD 'charscripts';
my $charscripts = charscripts();

use Unicode::UCD qw(charscript charinrange);
my $range = charscript($script);
print "looks like $script\n" if charinrange($range, $codepoint);

use Unicode::UCD qw(general_categories bidi_types);
my $categories = general_categories();
my $types = bidi_types();

use Unicode::UCD 'compexcl';
my $compexcl = compexcl($codepoint);

use Unicode::UCD 'namedseq';
my $namedseq = namedseq($named_sequence_name);

my $unicode_version = Unicode::UCD::UnicodeVersion();

my $convert_to_numeric =
           Unicode::UCD::num("\N{RUMI DIGIT ONE}\N{RUMI DIGIT TWO}");
```

DESCRIPTION

The Unicode::UCD module offers a series of functions that provide a simple interface to the Unicode Character Database.

code point argument

Some of the functions are called with a *code point argument*, which is either a decimal or a hexadecimal scalar designating a Unicode code point, or U+ followed by hexadecimals designating a

Unicode code point. In other words, if you want a code point to be interpreted as a hexadecimal number, you must prefix it with either `0x` or `U+`, because a string like e.g. `123` will be interpreted as a decimal code point. Note that the largest code point in Unicode is U+10FFFF. =cut

my $BLOCKSFH; my $VERSIONFH; my $CASEFOLDFH; my $CASESPECFH; my $NAMEDSEQFH;

sub openunicode { my ($rfh, @path) = @_; my $f; unless (defined $$rfh) { for my $d (@INC) { use File::Spec; $f = File::Spec->catfile($d, "unicore", @path); last if open($$rfh, $f); undef $f; } croak __PACKAGE__, ": failed to find ", File::Spec->catfile(@path), " in @INC" unless defined $f; } return $f; }

charinfo()

```
use Unicode::UCD 'charinfo';

my $charinfo = charinfo(0x41);
```

This returns information about the input *code point argument* as a reference to a hash of fields as defined by the Unicode standard. If the *code point argument* is not assigned in the standard (i.e., has the general category `Cn` meaning `Unassigned`) or is a non-character (meaning it is guaranteed to never be assigned in the standard), **undef** is returned.

Fields that aren't applicable to the particular code point argument exist in the returned hash, and are empty.

The keys in the hash with the meanings of their values are:

code

the input *code point argument* expressed in hexadecimal, with leading zeros added if necessary to make it contain at least four hexdigits

name

name of *code*, all IN UPPER CASE. Some control-type code points do not have names. This field will be empty for `Surrogate` and `Private Use` code points, and for the others without a name, it will contain a description enclosed in angle brackets, like `<control>`.

category

The short name of the general category of *code*. This will match one of the keys in the hash returned by *general_categories()*.

combining

the combining class number for *code* used in the Canonical Ordering Algorithm. For Unicode 5.1, this is described in Section 3.11 `Canonical Ordering Behavior` available at *http://www.unicode.org/versions/Unicode5.1.0/*

bidi

bidirectional type of *code*. This will match one of the keys in the hash returned by *bidi_types()*.

decomposition

is empty if *code* has no decomposition; or is one or more codes (separated by spaces) that taken in order represent a decomposition for *code*. Each has at least four hexdigits. The codes may be preceded by a word enclosed in angle brackets then a space, like `<compat>` , giving the type of decomposition

This decomposition may be an intermediate one whose components are also decomposable. Use *Unicode::Normalize* to get the final decomposition.

decimal

if *code* is a decimal digit this is its integer numeric value

digit

if *code* represents some other digit-like number, this is its integer numeric value

numeric

if *code* represents a whole or rational number, this is its numeric value. Rational values are expressed as a string like `1/4`.

mirrored

`Y` or `N` designating if *code* is mirrored in bidirectional text

unicode10

name of *code* in the Unicode 1.0 standard if one existed for this code point and is different from the current name

comment

As of Unicode 6.0, this is always empty.

upper

is empty if there is no single code point uppercase mapping for *code* (its uppercase mapping is itself); otherwise it is that mapping expressed as at least four hexdigits. (*casespec()* should be used in addition to **charinfo()** for case mappings when the calling program can cope with multiple code point mappings.)

lower

is empty if there is no single code point lowercase mapping for *code* (its lowercase mapping is itself); otherwise it is that mapping expressed as at least four hexdigits. (*casespec()* should be used in addition to **charinfo()** for case mappings when the calling program can cope with multiple code point mappings.)

title

is empty if there is no single code point titlecase mapping for *code* (its titlecase mapping is itself); otherwise it is that mapping expressed as at least four hexdigits. (*casespec()* should be used in addition to **charinfo()** for case mappings when the calling program can cope with multiple code point mappings.)

block

block *code* belongs to (used in `\p{Blk=...}`). See *Blocks versus Scripts*.

script

script *code* belongs to. See *Blocks versus Scripts*.

Note that you cannot do (de)composition and casing based solely on the *decomposition*, *combining*, *lower*, *upper*, and *title* fields; you will need also the *compexcl()*, and *casespec()* functions.

charblock()

```
use Unicode::UCD 'charblock';

my $charblock = charblock(0x41);
my $charblock = charblock(1234);
my $charblock = charblock(0x263a);
my $charblock = charblock("U+263a");

my $range     = charblock('Armenian');
```

With a *code point argument* charblock() returns the *block* the code point belongs to, e.g. `Basic Latin`. If the code point is unassigned, this returns the block it would belong to if it were assigned (which it may in future versions of the Unicode Standard).

See also *Blocks versus Scripts*.

If supplied with an argument that can't be a code point, charblock() tries to do the opposite and interpret the argument as a code point block. The return value is a *range*: an anonymous list of lists that contain *start-of-range*, *end-of-range* code point pairs. You can test whether a code point is in a range using the *charinrange()* function. If the argument is not a known code point block, **undef** is returned.

charscript()

```
use Unicode::UCD 'charscript';

my $charscript = charscript(0x41);
my $charscript = charscript(1234);
my $charscript = charscript("U+263a");

my $range      = charscript('Thai');
```

With a *code point argument* charscript() returns the *script* the code point belongs to, e.g. `Latin`, `Greek`, `Han`. If the code point is unassigned, it returns **undef**

If supplied with an argument that can't be a code point, charscript() tries to do the opposite and interpret the argument as a code point script. The return value is a *range*: an anonymous list of lists that contain *start-of-range*, *end-of-range* code point pairs. You can test whether a code point is in a range using the *charinrange()* function. If the argument is not a known code point script, **undef** is returned.

See also *Blocks versus Scripts*.

charblocks()

```
use Unicode::UCD 'charblocks';

my $charblocks = charblocks();
```

charblocks() returns a reference to a hash with the known block names as the keys, and the code point ranges (see *charblock()*) as the values.

See also *Blocks versus Scripts*.

charscripts()

```
use Unicode::UCD 'charscripts';

my $charscripts = charscripts();
```

charscripts() returns a reference to a hash with the known script names as the keys, and the code point ranges (see *charscript()*) as the values.

See also *Blocks versus Scripts*.

charinrange()

In addition to using the `\p{Blk=...}` and `\P{Blk=...}` constructs, you can also test whether a code point is in the *range* as returned by *charblock()* and *charscript()* or as the values of the hash returned by *charblocks()* and *charscripts()* by using charinrange():

```
use Unicode::UCD qw(charscript charinrange);

$range = charscript('Hiragana');
print "looks like hiragana\n" if charinrange($range, $codepoint);
```

general_categories()

```
use Unicode::UCD 'general_categories';

my $categories = general_categories();
```

This returns a reference to a hash which has short general category names (such as `Lu`, `Nd`, `Zs`, `S`) as keys and long names (such as `UppercaseLetter`, `DecimalNumber`, `SpaceSeparator`, `Symbol`) as values. The hash is reversible in case you need to go from the long names to the short names. The general category is the one returned from *charinfo()* under the `category` key.

bidi_types()

```
use Unicode::UCD 'bidi_types';

my $categories = bidi_types();
```

This returns a reference to a hash which has the short bidi (bidirectional) type names (such as `L`, `R`) as keys and long names (such as `Left-to-Right`, `Right-to-Left`) as values. The hash is reversible in case you need to go from the long names to the short names. The bidi type is the one returned from *charinfo()* under the `bidi` key. For the exact meaning of the various bidi classes the Unicode TR9 is recommended reading: *http://www.unicode.org/reports/tr9/* (as of Unicode 5.0.0)

compexcl()

```
use Unicode::UCD 'compexcl';

my $compexcl = compexcl(0x09dc);
```

This routine is included for backwards compatibility, but as of Perl 5.12, for most purposes it is probably more convenient to use one of the following instead:

```
my $compexcl = chr(0x09dc) =~ /\p{Comp_Ex};
my $compexcl = chr(0x09dc) =~ /\p{Full_Composition_Exclusion};
```

or even

```
my $compexcl = chr(0x09dc) =~ /\p{CE};
my $compexcl = chr(0x09dc) =~ /\p{Composition_Exclusion};
```

The first two forms return **true** if the *code point argument* should not be produced by composition normalization. The final two forms additionally require that this fact not otherwise be determinable from the Unicode data base for them to return **true**.

This routine behaves identically to the final two forms. That is, it does not return **true** if the code point has a decomposition consisting of another single code point, nor if its decomposition starts with a code point whose combining class is non-zero. Code points that meet either of these conditions should also not be produced by composition normalization, which is probably why you should use the `Full_Composition_Exclusion` property instead, as shown above.

The routine returns **false** otherwise.

casefold()

```
use Unicode::UCD 'casefold';

my $casefold = casefold(0xDF);
if (defined $casefold) {
    my @full_fold_hex = split / /, $casefold->{'full'};
    my $full_fold_string =
                join "", map {chr(hex($_))} @full_fold_hex;
    my @turkic_fold_hex =
                   split / /, ($casefold->{'turkic'} ne "")
                                    ? $casefold->{'turkic'}
                                    : $casefold->{'full'};
    my $turkic_fold_string =
                   join "", map {chr(hex($_))} @turkic_fold_hex;
}
if (defined $casefold && $casefold->{'simple'} ne "") {
    my $simple_fold_hex = $casefold->{'simple'};
    my $simple_fold_string = chr(hex($simple_fold_hex));
}
```

This returns the (almost) locale-independent case folding of the character specified by the *code point argument*.

If there is no case folding for that code point, **undef** is returned.

If there is a case folding for that code point, a reference to a hash with the following fields is returned:

code

the input *code point argument* expressed in hexadecimal, with leading zeros added if necessary to make it contain at least four hexdigits

full

one or more codes (separated by spaces) that taken in order give the code points for the case folding for *code*. Each has at least four hexdigits.

simple

is empty, or is exactly one code with at least four hexdigits which can be used as an alternative case folding when the calling program cannot cope with the fold being a sequence of multiple code points. If *full* is just one code point, then *simple* equals *full*. If there is no single code point folding defined for *code*, then *simple* is the empty string. Otherwise, it is an inferior, but still better-than-nothing alternative folding to *full*.

mapping

is the same as *simple* if *simple* is not empty, and it is the same as *full* otherwise. It can be considered to be the simplest possible folding for *code*. It is defined primarily for backwards compatibility.

status

is `C` (for `common`) if the best possible fold is a single code point (*simple* equals *full* equals *mapping*). It is `S` if there are distinct folds, *simple* and *full* (*mapping* equals *simple*). And it is `F` if there only a *full* fold (*mapping* equals *full*; *simple* is empty). Note that this describes the contents of *mapping*. It is defined primarily for backwards compatibility.

On versions 3.1 and earlier of Unicode, *status* can also be `I` which is the same as `C` but is a special case for dotted uppercase I and dotless lowercase i:

- If you use this `I` mapping, the result is case-insensitive, but dotless and dotted I's are

not distinguished

- If you exclude this `I` mapping, the result is not fully case-insensitive, but dotless and dotted I's are distinguished

turkic

contains any special folding for Turkic languages. For versions of Unicode starting with 3.2, this field is empty unless *code* has a different folding in Turkic languages, in which case it is one or more codes (separated by spaces) that taken in order give the code points for the case folding for *code* in those languages. Each code has at least four hexdigits. Note that this folding does not maintain canonical equivalence without additional processing.

For versions of Unicode 3.1 and earlier, this field is empty unless there is a special folding for Turkic languages, in which case *status* is `I`, and *mapping*, *full*, *simple*, and *turkic* are all equal.

Programs that want complete generality and the best folding results should use the folding contained in the *full* field. But note that the fold for some code points will be a sequence of multiple code points.

Programs that can't cope with the fold mapping being multiple code points can use the folding contained in the *simple* field, with the loss of some generality. In Unicode 5.1, about 7% of the defined foldings have no single code point folding.

The *mapping* and *status* fields are provided for backwards compatibility for existing programs. They contain the same values as in previous versions of this function.

Locale is not completely independent. The *turkic* field contains results to use when the locale is a Turkic language.

For more information about case mappings see *http://www.unicode.org/unicode/reports/tr21*

casespec()

```
use Unicode::UCD 'casespec';

my $casespec = casespec(0xFB00);
```

This returns the potentially locale-dependent case mappings of the *code point argument*. The mappings may be longer than a single code point (which the basic Unicode case mappings as returned by *charinfo()* never are).

If there are no case mappings for the *code point argument*, or if all three possible mappings (*lower*, *title* and *upper*) result in single code points and are locale independent and unconditional, **undef** is returned (which means that the case mappings, if any, for the code point are those returned by *charinfo()*).

Otherwise, a reference to a hash giving the mappings (or a reference to a hash of such hashes, explained below) is returned with the following keys and their meanings:

The keys in the bottom layer hash with the meanings of their values are:

code

the input *code point argument* expressed in hexadecimal, with leading zeros added if necessary to make it contain at least four hexdigits

lower

one or more codes (separated by spaces) that taken in order give the code points for the lower case of *code*. Each has at least four hexdigits.

title

one or more codes (separated by spaces) that taken in order give the code points for the title

case of *code*. Each has at least four hexdigits.

upper

one or more codes (separated by spaces) that taken in order give the code points for the upper case of *code*. Each has at least four hexdigits.

condition

the conditions for the mappings to be valid. If **undef**, the mappings are always valid. When defined, this field is a list of conditions, all of which must be true for the mappings to be valid. The list consists of one or more *locales* (see below) and/or *contexts* (explained in the next paragraph), separated by spaces. (Other than as used to separate elements, spaces are to be ignored.) Case distinctions in the condition list are not significant. Conditions preceded by "NON_" represent the negation of the condition.

A *context* is one of those defined in the Unicode standard. For Unicode 5.1, they are defined in Section 3.13 `Default Case Operations` available at *http://www.unicode.org/versions/Unicode5.1.0/*. These are for context-sensitive casing.

The hash described above is returned for locale-independent casing, where at least one of the mappings has length longer than one. If **undef** is returned, the code point may have mappings, but if so, all are length one, and are returned by *charinfo()*. Note that when this function does return a value, it will be for the complete set of mappings for a code point, even those whose length is one.

If there are additional casing rules that apply only in certain locales, an additional key for each will be defined in the returned hash. Each such key will be its locale name, defined as a 2-letter ISO 3166 country code, possibly followed by a "_" and a 2-letter ISO language code (possibly followed by a "_" and a variant code). You can find the lists of all possible locales, see *Locale::Country* and *Locale::Language*. (In Unicode 6.0, the only locales returned by this function are `lt`, `tr`, and `az`.)

Each locale key is a reference to a hash that has the form above, and gives the casing rules for that particular locale, which take precedence over the locale-independent ones when in that locale.

If the only casing for a code point is locale-dependent, then the returned hash will not have any of the base keys, like `code`, `upper`, etc., but will contain only locale keys.

For more information about case mappings see *http://www.unicode.org/unicode/reports/tr21/*

namedseq()

```
use Unicode::UCD 'namedseq';

my $namedseq = namedseq("KATAKANA LETTER AINU P");
my @namedseq = namedseq("KATAKANA LETTER AINU P");
my %namedseq = namedseq();
```

If used with a single argument in a scalar context, returns the string consisting of the code points of the named sequence, or **undef** if no named sequence by that name exists. If used with a single argument in a list context, it returns the list of the ordinals of the code points. If used with no arguments in a list context, returns a hash with the names of the named sequences as the keys and the named sequences as strings as the values. Otherwise, it returns **undef** or an empty list depending on the context.

This function only operates on officially approved (not provisional) named sequences.

Note that as of Perl 5.14, `\N{KATAKANA LETTER AINU P}` will insert the named sequence into double-quoted strings, and `charnames::string_vianame("KATAKANA LETTER AINU P")` will return the same string this function does, but will also operate on character names that aren't named sequences, without you having to know which are which. See *charnames*.

num

num returns the numeric value of the input Unicode string; or undef if it doesn't think the entire string has a completely valid, safe numeric value.

If the string is just one character in length, the Unicode numeric value is returned if it has one, or undef otherwise. Note that this need not be a whole number. num("\N{TIBETAN DIGIT HALF ZERO}"), for example returns -0.5.

If the string is more than one character, undef is returned unless all its characters are decimal digits (that is they would match \d+), from the same script. For example if you have an ASCII '0' and a Bengali '3', mixed together, they aren't considered a valid number, and undef is returned. A further restriction is that the digits all have to be of the same form. A half-width digit mixed with a full-width one will return undef. The Arabic script has two sets of digits; num will return undef unless all the digits in the string come from the same set.

num errs on the side of safety, and there may be valid strings of decimal digits that it doesn't recognize. Note that Unicode defines a number of "digit" characters that aren't "decimal digit" characters. "Decimal digits" have the property that they have a positional value, i.e., there is a units position, a 10's position, a 100's, etc, AND they are arranged in Unicode in blocks of 10 contiguous code points. The Chinese digits, for example, are not in such a contiguous block, and so Unicode doesn't view them as decimal digits, but merely digits, and so \d will not match them. A single-character string containing one of these digits will have its decimal value returned by num, but any longer string containing only these digits will return undef.

Strings of multiple sub- and superscripts are not recognized as numbers. You can use either of the compatibility decompositions in Unicode::Normalize to change these into digits, and then call num on the result.

Unicode::UCD::UnicodeVersion

This returns the version of the Unicode Character Database, in other words, the version of the Unicode standard the database implements. The version is a string of numbers delimited by dots ('.').

Blocks versus Scripts

The difference between a block and a script is that scripts are closer to the linguistic notion of a set of code points required to present languages, while block is more of an artifact of the Unicode code point numbering and separation into blocks of (mostly) 256 code points.

For example the Latin **script** is spread over several **blocks**, such as Basic Latin, Latin 1 Supplement, Latin Extended-A, and Latin Extended-B. On the other hand, the Latin script does not contain all the characters of the Basic Latin block (also known as ASCII): it includes only the letters, and not, for example, the digits or the punctuation.

For blocks see *http://www.unicode.org/Public/UNIDATA/Blocks.txt*

For scripts see UTR #24: *http://www.unicode.org/unicode/reports/tr24/*

Matching Scripts and Blocks

Scripts are matched with the regular-expression construct \p{...} (e.g. \p{Tibetan} matches characters of the Tibetan script), while \p{Blk=...} is used for blocks (e.g. \p{Blk=Tibetan} matches any of the 256 code points in the Tibetan block).

Implementation Note

The first use of charinfo() opens a read-only filehandle to the Unicode Character Database (the database is included in the Perl distribution). The filehandle is then kept open for further queries. In other words, if you are wondering where one of your filehandles went, that's where.

BUGS

Does not yet support EBCDIC platforms.

AUTHOR

Jarkko Hietaniemi