NAME

charnames - access to Unicode character names and named character sequences; also define character names

SYNOPSIS

```
use charnames ':full';
print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";
print "\N{LATIN CAPITAL LETTER E WITH VERTICAL LINE BELOW}",
      " is an officially named sequence of two Unicode characters\n";

use charnames ':loose';
print "\N{Greek small-letter  sigma}",
       "can be used to ignore case, underscores, most blanks,"
       "and when you aren't sure if the official name has hyphens\n";

use charnames ':short';
print "\N{greek:Sigma} is an upper-case sigma.\n";

use charnames qw(cyrillic greek);
print "\N{sigma} is Greek sigma, and \N{be} is Cyrillic b.\n";

use charnames ":full", ":alias" => {
  e_ACUTE => "LATIN SMALL LETTER E WITH ACUTE",
  mychar => 0xE8000,  # Private use area
};
print "\N{e_ACUTE} is a small letter e with an acute.\n";
print "\N{mychar} allows me to name private use characters.\n";

use charnames ();
print charnames::viacode(0x1234); # prints "ETHIOPIC SYLLABLE SEE"
printf "%04X", charnames::vianame("GOTHIC LETTER AHSA"); # prints
                                                         # "10330"
print charnames::vianame("LATIN CAPITAL LETTER A"); # prints 65 on
                                                    # ASCII platforms;
                                                    # 193 on EBCDIC
print charnames::string_vianame("LATIN CAPITAL LETTER A"); # prints "A"
```

DESCRIPTION

Pragma `use charnames` is used to gain access to the names of the Unicode characters and named character sequences, and to allow you to define your own character and character sequence names.

All forms of the pragma enable use of the following 3 functions:

- *charnames::string_vianame(name)* for run-time lookup of a either a character name or a named character sequence, returning its string representation
- *charnames::vianame(name)* for run-time lookup of a character name (but not a named character sequence) to get its ordinal value (code point)
- *charnames::viacode(code)* for run-time lookup of a code point to get its Unicode name.

Starting in Perl v5.16, any occurrence of `\N{`*CHARNAME*`}` sequences in a double-quotish string automatically loads this module with arguments `:full` and `:short` (described below) if it hasn't already been loaded with different arguments, in order to compile the named Unicode character into position in the string. Prior to v5.16, an explicit `use charnames` was required to enable this usage.

(However, prior to v5.16, the form "`use charnames ();`" did not enable `\N{`*CHARNAME*`}`.)

Note that `\N{U+...}`, where the *...* is a hexadecimal number, also inserts a character into a string. The character it inserts is the one whose code point (ordinal value) is equal to the number. For example, "`\N{U+263a}`" is the Unicode (white background, black foreground) smiley face equivalent to "`\N{WHITE SMILING FACE}`". Also note, `\N{...}` can mean a regex quantifier instead of a character name, when the *...* is a number (or comma separated pair of numbers (see *"QUANTIFIERS" in perlreref*), and is not related to this pragma.

The `charnames` pragma supports arguments `:full`, `:loose`, `:short`, script names and *customized aliases*.

If `:full` is present, for expansion of `\N{`*CHARNAME*`}`, the string *CHARNAME* is first looked up in the list of standard Unicode character names.

`:loose` is a variant of `:full` which allows *CHARNAME* to be less precisely specified. Details are in *LOOSE MATCHES*.

If `:short` is present, and *CHARNAME* has the form *SCRIPT*`:`*CNAME*, then *CNAME* is looked up as a letter in script *SCRIPT*, as described in the next paragraph. Or, if `use charnames` is used with script name arguments, then for `\N{`*CHARNAME*`}` the name *CHARNAME* is looked up as a letter in the given scripts (in the specified order). Customized aliases can override these, and are explained in *CUSTOM ALIASES*.

For lookup of *CHARNAME* inside a given script *SCRIPTNAME*, this pragma looks in the table of standard Unicode names for the names

```
SCRIPTNAME CAPITAL LETTER CHARNAME
SCRIPTNAME SMALL LETTER CHARNAME
SCRIPTNAME LETTER CHARNAME
```

If *CHARNAME* is all lowercase, then the `CAPITAL` variant is ignored, otherwise the `SMALL` variant is ignored, and both *CHARNAME* and *SCRIPTNAME* are converted to all uppercase for look-up. Other than that, both of them follow *loose* rules if `:loose` is also specified; strict otherwise.

Note that `\N{...}` is compile-time; it's a special form of string constant used inside double-quotish strings; this means that you cannot use variables inside the `\N{...}`. If you want similar run-time functionality, use *charnames::string_vianame()*.

Since Unicode 6.0, it is deprecated to use `BELL`. Instead use `ALERT` (but `BEL` will continue to work).

If the input name is unknown, `\N{NAME}` raises a warning and substitutes the Unicode REPLACEMENT CHARACTER (U+FFFD).

For `\N{NAME}`, it is a fatal error if `use bytes` is in effect and the input name is that of a character that won't fit into a byte (i.e., whose ordinal is above 255).

Otherwise, any string that includes a `\N{`*charname*`}` or `\N{U+`*code point*`}` will automatically have Unicode semantics (see *"Byte and Character Semantics" in perlunicode*).

LOOSE MATCHES

By specifying `:loose`, Unicode's *loose character name matching* rules are selected instead of the strict exact match used otherwise. That means that *CHARNAME* doesn't have to be so precisely specified. Upper/lower case doesn't matter (except with scripts as mentioned above), nor do any underscores, and the only hyphens that matter are those at the beginning or end of a word in the name (with one exception: the hyphen in U+1180 `HANGUL JUNGSEONG O-E` does matter). Also, blanks not adjacent to hyphens don't matter. The official Unicode names are quite variable as to where they use hyphens versus spaces to separate word-like units, and this option allows you to not have to care as much. The reason non-medial hyphens matter is because of cases like U+0F60 `TIBETAN LETTER -A` versus U+0F68 `TIBETAN LETTER A`. The hyphen here is significant, as is

the space before it, and so both must be included.

`:loose` slows down look-ups by a factor of 2 to 3 versus `:full`, but the trade-off may be worth it to you. Each individual look-up takes very little time, and the results are cached, so the speed difference would become a factor only in programs that do look-ups of many different spellings, and probably only when those look-ups are through vianame() and string_vianame(), since `\N{...}` look-ups are done at compile time.

ALIASES

Starting in Unicode 6.1 and Perl v5.16, Unicode defines many abbreviations and names that were formerly Perl extensions, and some additional ones that Perl did not previously accept. The list is getting too long to reproduce here, but you can get the complete list from the Unicode web site: *http://www.unicode.org/Public/UNIDATA/NameAliases.txt*.

Earlier versions of Perl accepted almost all the 6.1 names. These were most extensively documented in the v5.14 version of this pod: *http://perldoc.perl.org/5.14.0/charnames.html#ALIASES*.

CUSTOM ALIASES

You can add customized aliases to standard (`:full`) Unicode naming conventions. The aliases override any standard definitions, so, if you're twisted enough, you can change `"\N{LATIN CAPITAL LETTER A}"` to mean `"B"`, etc.

Note that an alias should not be something that is a legal curly brace-enclosed quantifier (see *"QUANTIFIERS" in perlreref*). For example `\N{123}` means to match 123 non-newline characters, and is not treated as a charnames alias. Aliases are discouraged from beginning with anything other than an alphabetic character and from containing anything other than alphanumerics, spaces, dashes, parentheses, and underscores. Currently they must be ASCII.

An alias can map to either an official Unicode character name (not a loose matched name) or to a numeric code point (ordinal). The latter is useful for assigning names to code points in Unicode private use areas such as U+E800 through U+F8FF. A numeric code point must be a non-negative integer or a string beginning with `"U+"` or `"0x"` with the remainder considered to be a hexadecimal integer. A literal numeric constant must be unsigned; it will be interpreted as hex if it has a leading zero or contains non-decimal hex digits; otherwise it will be interpreted as decimal.

Aliases are added either by the use of anonymous hashes:

```
use charnames ":alias" => {
    e_ACUTE => "LATIN SMALL LETTER E WITH ACUTE",
    mychar1 => 0xE8000,
    };
my $str = "\N{e_ACUTE}";
```

or by using a file containing aliases:

```
use charnames ":alias" => "pro";
```

This will try to read `"unicore/pro_alias.pl"` from the `@INC` path. This file should return a list in plain perl:

```
(
A_GRAVE         => "LATIN CAPITAL LETTER A WITH GRAVE",
A_CIRCUM        => "LATIN CAPITAL LETTER A WITH CIRCUMFLEX",
A_DIAERES       => "LATIN CAPITAL LETTER A WITH DIAERESIS",
A_TILDE         => "LATIN CAPITAL LETTER A WITH TILDE",
A_BREVE         => "LATIN CAPITAL LETTER A WITH BREVE",
A_RING          => "LATIN CAPITAL LETTER A WITH RING ABOVE",
A_MACRON        => "LATIN CAPITAL LETTER A WITH MACRON",
```

```
    mychar2         => "U+E8001",
    );
```

Both these methods insert `":full"` automatically as the first argument (if no other argument is given), and you can give the `":full"` explicitly as well, like

```
    use charnames ":full", ":alias" => "pro";
```

`":loose"` has no effect with these. Input names must match exactly, using `":full"` rules.

Also, both these methods currently allow only single characters to be named. To name a sequence of characters, use a *custom translator* (described below).

charnames::string_vianame(name)

This is a runtime equivalent to `\N{...}`. *name* can be any expression that evaluates to a name accepted by `\N{...}` under the *`:full` option* to `charnames`. In addition, any other options for the controlling `"use charnames"` in the same scope apply, like `:loose` or any *script list, `:short` option*, or *custom aliases* you may have defined.

The only difference is that if the input name is unknown, `string_vianame` returns `undef` instead of the REPLACEMENT CHARACTER and does not raise a warning message.

charnames::vianame(name)

This is similar to `string_vianame`. The main difference is that under most circumstances, vianame returns an ordinal code point, whereas `string_vianame` returns a string. For example,

```
   printf "U+%04X", charnames::vianame("FOUR TEARDROP-SPOKED ASTERISK");
```

prints "U+2722".

This leads to the other two differences. Since a single code point is returned, the function can't handle named character sequences, as these are composed of multiple characters (it returns `undef` for these. And, the code point can be that of any character, even ones that aren't legal under the `use bytes` pragma,

See *BUGS* for the circumstances in which the behavior differs from that described above.

charnames::viacode(code)

Returns the full name of the character indicated by the numeric code. For example,

```
    print charnames::viacode(0x2722);
```

prints "FOUR TEARDROP-SPOKED ASTERISK".

The name returned is the "best" (defined below) official name or alias for the code point, if available; otherwise your custom alias for it, if defined; otherwise `undef`. This means that your alias will only be returned for code points that don't have an official Unicode name (nor alias) such as private use code points.

If you define more than one name for the code point, it is indeterminate which one will be returned.

As mentioned, the function returns `undef` if no name is known for the code point. In Unicode the proper name of these is the empty string, which `undef` stringifies to. (If you ask for a code point past the legal Unicode maximum of U+10FFFF that you haven't assigned an alias to, you get `undef` plus a warning.)

The input number must be a non-negative integer, or a string beginning with `"U+"` or `"0x"` with the remainder considered to be a hexadecimal integer. A literal numeric constant must be unsigned; it will

be interpreted as hex if it has a leading zero or contains non-decimal hex digits; otherwise it will be interpreted as decimal.

As mentioned above under *ALIASES*, Unicode 6.1 defines extra names (synonyms or aliases) for some code points, most of which were already available as Perl extensions. All these are accepted by `\N{...}` and the other functions in this module, but `viacode` has to choose which one name to return for a given input code point, so it returns the "best" name. To understand how this works, it is helpful to know more about the Unicode name properties. All code points actually have only a single name, which (starting in Unicode 2.0) can never change once a character has been assigned to the code point. But mistakes have been made in assigning names, for example sometimes a clerical error was made during the publishing of the Standard which caused words to be misspelled, and there was no way to correct those. The Name_Alias property was eventually created to handle these situations. If a name was wrong, a corrected synonym would be published for it, using Name_Alias. `viacode` will return that corrected synonym as the "best" name for a code point. (It is even possible, though it hasn't happened yet, that the correction itself will need to be corrected, and so another Name_Alias can be created for that code point; `viacode` will return the most recent correction.)

The Unicode name for each of the control characters (such as LINE FEED) is the empty string. However almost all had names assigned by other standards, such as the ASCII Standard, or were in common use. `viacode` returns these names as the "best" ones available. Unicode 6.1 has created Name_Aliases for each of them, including alternate names, like NEW LINE. `viacode` uses the original name, "LINE FEED" in preference to the alternate. Similarly the name returned for U+FEFF is "ZERO WIDTH NO-BREAK SPACE", not "BYTE ORDER MARK".

Until Unicode 6.1, the 4 control characters U+0080, U+0081, U+0084, and U+0099 did not have names nor aliases. To preserve backwards compatibility, any alias you define for these code points will be returned by this function, in preference to the official name.

Some code points also have abbreviated names, such as "LF" or "NL". `viacode` never returns these.

Because a name correction may be added in future Unicode releases, the name that `viacode` returns may change as a result. This is a rare event, but it does happen.

CUSTOM TRANSLATORS

The mechanism of translation of `\N{...}` escapes is general and not hardwired into *charnames.pm*. A module can install custom translations (inside the scope which `use`s the module) with the following magic incantation:

```
sub import {
    shift;
    $^H{charnames} = \&translator;
}
```

Here translator() is a subroutine which takes *CHARNAME* as an argument, and returns text to insert into the string instead of the `\N{`*CHARNAME*`}` escape.

This is the only way you can create a custom named sequence of code points.

Since the text to insert should be different in `bytes` mode and out of it, the function should check the current state of `bytes`-flag as in:

```
use bytes ();                       # for $bytes::hint_bits
sub translator {
    if ($^H & $bytes::hint_bits) {
        return bytes_translator(@_);
    }
    else {
        return utf8_translator(@_);
    }
```

```
}
```

See *CUSTOM ALIASES* above for restrictions on *CHARNAME*.

Of course, `vianame`, `viacode`, and `string_vianame` would need to be overridden as well.

BUGS

vianame() normally returns an ordinal code point, but when the input name is of the form `U+...`, it returns a chr instead. In this case, if `use bytes` is in effect and the character won't fit into a byte, it returns `undef` and raises a warning.

Names must be ASCII characters only, which means that you are out of luck if you want to create aliases in a language where some or all the characters of the desired aliases are non-ASCII.

Since evaluation of the translation function (see *CUSTOM TRANSLATORS*) happens in the middle of compilation (of a string literal), the translation function should not do any `eval`s or `require`s. This restriction should be lifted (but is low priority) in a future version of Perl.