Package ‘qqid’

April 29, 2019

Type Package

Title Generation and Support of QQIDs - A Human-Compatible Representation of 128-bit Numbers

Version 1.0.3

Date 2019-04-25

Description The string ``bird.carp.7TsBWtwqtKAeCTNk8f'' is a ``QQID'': a representation of a 128-bit number, constructed from two ``cues'' of short, common, English words, and Base64 encoded characters. The primary intended use of QQIDs is as random unique identifiers, e.g. database keys like the ``UUIDs'' defined in the RFC 4122 Internet standard. QQIDs can be identically interconverted with UUIDs, IPv6 addresses, MD5 hashes etc., and are suitable for a host of applications in which identifiers are read by humans. They are compact, can safely be transmitted in binary and text form, can be used as components of URLs, and it can be established at a glance whether two QQIDs are different or potentially identical. The qqid package contains functions to retrieve true, quantum-random QQIDs, to generate pseudo- random QQIDs, to validate them, and to interconvert them with other 128-bit number representations.

License MIT + file LICENSE

Language en-GB

Encoding UTF-8

LazyData true

Imports qrandom

Suggests covr, digest, testthat

RoxygenNote 6.1.1

URL https://github.com/hyginn/qqid

BugReports https://github.com/hyginn/qqid/issues

NeedsCompilation no

Author Boris Steipe [aut, cre] (<https://orcid.org/0000-0002-1134-6758>)

Maintainer Boris Steipe <boris.steipe@utoronto.ca>

Repository CRAN

Date/Publication 2019-04-29 09:40:03 UTC

R topics documented:

`is.QQID` *is.QQID*

Description

`is.QQID` tests whether the function argument is a vector of valid QQIDs.

Usage

```
is.QQID(s, na.map = FALSE)
```

Arguments

`s`	(character) a vector of strings to check.
`na.map`	(logical) replace NA with `FALSE` (default), `NA`, or `TRUE`

Details

`is.QQID` accepts a vector of strings and returns a logical vector of the same length, `TRUE` for every element of the input that is a valid QQID. `NA` values are mapped to `FALSE` (default) or can be replaced with `NA` to preserve them. Note: arguments passed to `na.map` are implicitly converted to type logical. A valid QQID has the form: `"qqqq.qqqq.BBBBBBBBBBBBBBBBBB"` where each `"qqqq"` is a Q-word (cf. `qMap()`) encoding 10 bit, and `"B"` is a Base64 character encoding 6 bits each for `10 + 10 + (18 * 6) == 128` bits. The two Q-words of the QQID are separated by a "." which does not appear in UUIDs nor in the Base64 alphabet and thus the Q-words bead, beef, dead, deaf, deed, face, and fade can be easily distinguished from 4 digit hexadecimal numbers in a QQID or a Base64 encoded number.

Value

(logical) a vector of the same length as the input, TRUE for every element of the input that is a valid QQID.

Author(s)

See Also

is.xlt() to check UUIDs, MD5s, IPv6 addresses and other hexlets.

Examples

```
# check one invalid QQID
is.QQID("spur.ious.oversimplification")  # FALSE: "ious" is not a Q-word

# check one valid QQID
is.QQID("base.less.Anthr0p0centricity")  # TRUE, perhaps regrettably so

# check two valid QQIDs
is.QQID(QQIDexample(1:2))   # TRUE  TRUE

# convert a UUID and check it
is.QQID(xlt2qq("0c460ed3-b015-adc2-ab4a-01e093364f1f"))   # TRUE

# check a valid QQID, not a QQID, and an NA. Map NA to NA, not to FALSE.
is.QQID(c(QQIDexample(3), "meh", NA), na.map = NA)   # TRUE  FALSE  NA
```

---

is.xlt *is.xlt*

Description

is.xlt tests whether the function argument is a vector of 32 digit hexadecimal numbers (a "hexlet").

Usage

```
is.xlt(s, na.map = FALSE)
```

Arguments

s	(character) a vector of strings to check.
na.map	(logical) replace NA with FALSE (default), NA, or TRUE

Details

`is.xlt` accepts a vector of strings and returns a logical vector of the same length, `TRUE` for every element of the input that matches the regular expression `"^[0-9a-f]{32}$"`. `NA` values are mapped to `FALSE` (default) or can be replaced with `NA` to preserve them. Note: arguments passed to `na.map` are implicitly converted to type logical.

Value

(logical) a vector of the same length as the input, `TRUE` for every element of the input that is a valid UUID, `FALSE` for every element that is not, and `NA` or `FALSE` for every `NA` input.

Author(s)

(c) 2019 Boris Steipe, licensed under MIT (see file `LICENSE` in this package).

See Also

`is.QQID()` to check QQIDs.

Examples

```
# check the example hexlet formats
is.xlt(xltIDexample())              # TRUE  TRUE  TRUE  TRUE  TRUE

# check one invalid format
is.xlt("2.718281828459045235360287471353")     # FALSE

# check a valid hexlet, an invalid hexlet, and an NA.
# Map input NA to NA, not to FALSE.
is.xlt(c("0c46:0ed3:b015:adc2:ab4a:01e0:9336:4f1f", # IPv6
         "c46:ed3:b015:adc2:ab4a:1e0:9336:4f1f",    # IPv6 abbreviated format
         NA), na.map = NA)                          # TRUE  FALSE  NA
```

---

qMap *qMap*

Description

`qMap` maps numbers to Q-words, or Q-words to their index in (0, 1023).

Usage

```
qMap(x)
```

Arguments

`x`	(character or numeric) A vector.

Value

(numeric or character) A vector of indices, Q-words, or NA of the same length as the input.

Description

`qMap` accepts strings that are matched to Q-word indices or NA, or numbers that are matched to Q-words or NA. The returned vector has the same length as the input. Numbers that are not in (0, 1023) return NA. Strings that are not a Q-word return NA. `qMap(0)` is "aims", `qMap(1023)` is "zone". Note: the first Q-word has index 0, since it represents the bit-pattern "0000000000".

Q-Words

A table of 1,024 four-letter words is encoded in this function. Four-letter English words were chosen and manually refined to yield short, unique labels that:

- are monosyllabic,
- are easy to spell and pronounce,
- are individually not offensive,
- are unlikely to be offensive in random combination,
- are in common use,
- avoid homophones and consonant clusters,
- do not contain jargon, intentional misspellings, acronyms or overly specialized technical or sports terms.

The table is alphabetically sorted.

Author(s)

Examples

```
# qMap a number
qMap(313)                                   # "gift"
# qMap four words, three can be matched.
qMap(c("three", "free", "cold", "beer"))    # NA 287  125  34
# return the entire QQ table
x <- qMap(0:1023)
```

qq2uu *qq2uu*

Description

qq2uu converts a vector of QQIDs to UUIDs.

Usage

```
qq2uu(qq)
```

Arguments

qq	(character) a vector of QQIDs

Value

(character) a vector of UUIDs

QQIDs

QQIDs are specially formatted 128-bit numbers (hexlets), just like UUIDs. See `xlt2qq()` for the the motivation of mapping UUIDs to QQIDs and details on how QQIDs are structured. qq2uu reverses the mapping exactly to recover the original UUID.

Process

To convert a QQID to a UUID, the two "Q-words" that head the QQID are mapped to their index in the 0:1023 Q-word vector (cf. `qMap()`), and the indices are converted to two ten bit numbers. These twenty bits are expressed as a five-digit hexadecimal number which replaces the two Q-words to recover the UUID. For details on UUID format see `is.xlt()`. The remaining 18 Base64 encoded characters are converted to their corresponding 27 hex digits via an intermediate mapping to bit-patterns.

Endianness

The qqid package uses its own functions to convert to and from bits, and is not affected by big-endian vs. little-endian processor architecture or variant byte order. All numbers are interpreted to have their lowest order digits on the right.

Author(s)

See Also

`xlt2qq()` to convert a vector of UUIDs, IPv6 addresses or other hexlets to QQIDs.

Examples

```
# Convert three example QQIDs and one NA to the corresponding UUIDs
qq2uu( c(QQIDexample(c(1, 3, 5)), NA) )

# forward and back again
myID <- "bird.carp.7TsBWtwqtKAeCTNk8f"
myID == xlt2qq(qq2uu(myID))             # TRUE

# Confirm that example QQID No. 3 is formatted correctly as a UUID
qq2uu( QQIDexample(3) ) == xltIDexample("UUID")  # TRUE
```

`QQIDexample` *QQIDexample*

Description

`QQIDexample` returns synthetic, valid QQIDs for testing and development. The synthetic examples are easy to distinguish from "real" IDs to prevent their accidental use in an application.

Usage

```
QQIDexample(sel = 1:5)
```

Arguments

`sel`	(numeric, or logical) a subsetting vector

Details

The function stores five artificial QQIDs. Input is an index vector that specifies which QQIDs to return. More than five IDs can be requested by applying the usual subsetting rules. The QQIDs represent the exact same numbers provided by `xltIDexample()`. However the `qqid` package provides only format conversion to UUID at this time, so the reverse comparison will only succeed with `xltIDexample("UUID")`.

Value

(character) a vector of QQIDs

Author(s)

See Also

`xltIDexample()` Returns five 128-bit "hexlets", formatted as Md5, hex-number, UUID, and IPv6.

Examples

```
QQIDexample()                                  # the five stored QQIDs
QQIDexample(2:3)                               # two QQIDS
QQIDexample(c(TRUE, FALSE))                    # vector recycling
QQIDexample(sample(1:5, 17, replace = TRUE))   # seventeen in random order
QQIDexample() == xlt2qq(xltIDexample())        # TRUE TRUE TRUE TRUE TRUE
```

`qQQIDfactory` *qQQIDfactory*

Description

`qQQIDfactory` returns a closure (a function with an associated environment) that retrieves, caches, and returns quantum-random QQIDs.

Usage

```
qQQIDfactory(nBatch = 1023)
```

Arguments

`nBatch`	(numeric) The batch size requested from the ANU server. The default 1023 does not normally need to be changed. (Larger batches take more time to process, smaller batches offer no speed benefit). It might be increased e.g. for storing a large cache in case an interruption to Internet connectivity is anticipated. Values < 1 and > 1e05 will cause an error in `qrandom::qUUID()`.

Details

In what follows we will call the closure that is produced by `qQQIDfactory` "`qQQID()`" (even though you could assign it to a different name). The function factory `qQQIDfactory` and the `qQQID()` closure address a latency problem that arises when we retrieve true random numbers from the ANU Quantum Random Number Generator at the Australian National University. `qQQIDfactory` produced closures maintain a local cache of true random QQIDs, which can be accessed without latency. In a design use case, `qQQID()` could be produced in an R session startup script. From that point on, latency in generating true random QQIDs is only experienced occasionally when the cache needs to be replenished.

Value

`qQQIDfactory` returns a closure that takes the following arguments:

- `n` (numeric) (default: 1; in interval: (1, 1e+05)): the number of true random QQIDs to return.
- `inspectOnly` (logical) (default: `FALSE`). If `TRUE`, the requested number of QQIDs are only printed as a side-effect, the function returns `NULL` invisibly. This is useful to inspect the first `n` elements of the cache without changing the cache, while making it sufficiently hard to accidentally assign and reuse QQIDs.

If no connection to ANU can be established and the initial cache cannot be filled, `qQQIDfactory` returns `NULL` invisibly.

Usage

`qQQIDfactory`produces a closure which is meant to be assigned to a variable and called by that variable name (see examples). While any legal variable name can be used, assigning to "`qQQID`" is recommended. Producing the closure requires an Internet connection to first fill the closure's cache of QQID values. Once the cache is filled, no Internet connection is required until the cache is depleted and the closure attempts to replenish it. If replenishing the cache fails because of a failure to open a connection to the ANU server, an informative message is printed and the closure returns `NULL` invisibly. If there are remaining QQIDs in the cache, these can be retrieved by requesting no more than the number that currently exist in the cache. The closure either returns the requested number of QQIDs, or NULL, but it will never return fewer than the requested number of QQIDs, since implicit recycling of shorter vectors would be a critical failure mode for a function that is expected to return unique identifiers.

Properties of the produced QQIDs

Internally, `qQQID` converts true random UUIDs obtained from `qrandom::qUUID`. These UUIDs are RFC 4122 compliant and contain a six-bit version code, and 122 random bits. Such RFC compliant QQIDs are drawn from $2^{122} \approx 5.3 \times 10^{36}$ possibilities, whereas totally random 128-bit numbers are drawn from a $\approx 3.4 \times 10^{38}$ number space. The 50% collision probability of random 122-bit numbers is $\approx 2.7 \times 10^{18}$ numbers, while for 128-bit numbers it is $\approx 2.2 \times 10^{19}$.

qQQIDs vs. rngQQIDs

Whether to use true random or pseudo-random QQIDs is a tradeoff between speed and safety. The ANU quantum random number server can have considerable latency (a problem that `qQQIDfactory` addresses through caching), but pseudo-random numbers may not be sufficiently collision-safe for use cases that depend on the uniqueness of the resulting numbers: while the RNGs provided by R are very good, all RNGs potentially suffer from the possibility of an **initialization** collision, i.e. when two runs of the RNG are accidentally initialized with the same seed, due to an improper and/or unrecognized use of `set.seed()` in another function, script or package, or due to the limited randomness of time- and machine-state based seeds. This is not a problem for long runs of key-generation on a single machine, but it may be an issue for the decentralized generation of random unique keys, which is the design use case of `qqid`. The only way to prevent this with certainty is to use true random keys (as provided with this function). True random qQQIDs have a 50% collision probability in $\approx 2.7 \times 10^{18}$ keys, and this is the same at all times, regardless of the state of the requesting machine. Thus unless throughput of keys is a critical concern, it is advisable to use true random QQIDs from a `qQQIDfactory` closure over those returned by a `rngQQID()` process, or at least to initialize the RNG with a true random seed (the default option for `rngQQID()`).

Caching QQIDs to avoid latency

The ANU server produces true random numbers from quantum fluctuations of the vacuum. The high latency of requests for quantum random numbers from the ANU server - between 6 to 10 seconds per request - is practically independent of the size of the request's payload. This suggests a strategy to serve keys from a local cache. R provides an elegant way of doing this by defining

functions as closures - i.e. along with their own environment. This function environment allows to maintain state between function calls. `qQQIDfactory` retrieves 1023 qQQIDs and caches them in the environment of `qQQID`. The `qQQID` closure replenishes the cache if it does not contain at least the requested number of QQIDs, then it unshifts the QQIDs from the cache, and returns them. The user experience is that `qQQID` is responsive, and latency arises only occasionally when the cache is replenished.

Warning - parallelization

If you are executing code in parallel on separate processors, you must make sure that every task uses its own, independent copy of `qQQID()` and not a copy of an instance of the closure - such copies would all contain the same cache! Given the relatively high latency of cache replenishment, it is likely that an approach that uses `rngQQID(N, method = "q")`, is more promising.

Disclaimer and caution

Although this function has been written and tested with care, no suitability for any particular purpose, in particular no suitability for high-value transactions, for applications whose failure could endanger life or property, or for cryptography is claimed. The source code is published in full and it is up to the user to audit and adapt the code for their own purposes and needs.

Author(s)

See Also

`rngQQID()` to generate QQIDs via the inbuilt RNG.

Examples

```
## Not run:
# prepare a qQQID function and use it to retrieve three true random  QQIDs
qQQID <- qQQIDfactory()
qQQID(3)

# Use the function to return 10 UUIDs from the same cache
qq2uu(qQQID(10))

# inspect the first 5 QQIDs on the cache ...
qQQID(5, inspectOnly = TRUE)

# ... retrieve four of the QQIDs (they are identical
# to the first four we just inspected) ...
qQQID(4)

# ... show that the four keys are gone from the cache which now
# begins with the fifth QQID we saw before.
qQQID(5, inspectOnly = TRUE)
```

```
## End(Not run)
```

---

rngQQID *rngQQID*

Description

rngQQID uses R's random number generator to generate a vector of pseudo-random QQIDs.

Usage

```
rngQQID(n = 1, method = "q", RFC4122compliant = TRUE)
```

Arguments

n	(integer) The number of pseudo-random QQIDs to return. Default is 1.
method	(character) Which random seed method to use. Default is "q", a true random number seed. Other options are "R" (R's inbuilt RNG initialization), "n" (no initialization, a reproducible random seed can be set prior to this call if one clearly understands the risks), "t" (test of error handling).
RFC4122compliant	(logical) whether to base the QQID on version-stamped 128-bit numbers with 122 random bits, compliant with RFC 4122 (default), or to return QQIDs based on 128 random bits.

Details

The function rngQQID generates a vector of n pseudo-random QQIDs using R's inbuilt random number generator. Internally, the QQIDs are constructed from randomly sampled individual bits, thus the resulting QQIDs do not suffer from distributional issues that arise from mapping large floating point numbers to a continuous range of integers. The function takes care not to change the global state of the RNG in .Random.seed.

Value

(character) a vector of n QQIDs

qQQIDs vs. rngQQIDs

Whether to use true random or pseudo-random QQIDs is a tradeoff between speed and safety. The ANU quantum random number server can have considerable latency (a problem that qQQIDfactory addresses through caching), but pseudo-random numbers may not be sufficiently collision-safe for use cases that depend on the uniqueness of the resulting numbers: while the RNGs provided by R are very good, all RNGs potentially suffer from the possibility of an **initialization** collision, i.e. when two runs of the RNG are accidentally initialized with the same seed, due to an improper and/or unrecognized use of set.seed() in another function, script or package, or due to the limited

randomness of time- and machine-state based seeds. This is not a problem for long runs of key-generation on a single machine, but it may be an issue for the decentralized generation of random unique keys, which is the design use case of `qqid`. The only way to prevent this with certainty is to use true random keys (as provided with this function). True random qQQIDs have a 50% collision probability in $\approx 2.7 \times 10^{18}$ keys, and this is the same at all times, regardless of the state of the requesting machine. Thus unless throughput of keys is a critical concern, it is advisable to use true random QQIDs from a `qQQIDfactory` closure over those returned by a `rngQQID()` process, or at least to initialize the RNG with a true random seed (method "q", the default option for `rngQQID`).

128 vs. 122 bit random

By default, QQIDs produced with `rngQQID()` can be converted to RFC 4122 compliant UUIDs. These use 6 bits to identify the method of UUID generation and thus contain only 122 random bits. It is possible to obtain 128-bit random QQIDs from `rngQQID()`, by setting the parameter `RFC4122compliant` to `FALSE`. This increases the number space from $2^{122} \approx 5.3 \times 10^{36}$ to $2^{128} \approx 3.4 \times 10^{38}$ at the cost of no longer being compliant with the UUID standard.

Random seeds

The function supports three methods to seed R's RNG. The default method is `"q"` and uses a true random seed retrieved from the ANU quantum random number server. An alternative method is `"r"`, which uses R's inbuilt random initialization (cf. the behaviour of `set.seed(NULL)` in the set.seed() documentation). Finally, the function can be run without a random seed with `"n"`, which allows either to define one's own sane RNG initialization, or use a specific seed for reproducible randomization - assuming that the risks are clearly understood. In all cases, the current state of R's RNG is saved and restored upon exit, even if the function exits with an error. For testing purposes, saving the RNG state can be demonstrated with method `"t"` which does not change the global random seed, creates exactly one random 128-bit number internally, and then throws an error to exit the function which should restore `.Random.seed`.

Warning - parallelization

If you are executing code in parallel on separate processors, you must make sure that every task uses its own, separately initialized state of the global variable `.Random.seed` and not a copy of a single instance of the global environment - such copies will produce exactly identical QQIDs unless reinitialized after the task is deployed.

Disclaimer and caution

Although this function has been written and tested with care, no suitability for any particular purpose, in particular no suitability for high-value transactions, for applications whose failure could endanger life or property, or for cryptography is claimed. The source code is published in full and it is up to the user to audit and adapt the code for their own purposes and needs.

Author(s)

See Also

qQQIDfactory() to create a closure that returns cached, true random QQIDs.

Examples

```
## Not run:

# initialize the RNG with a true random number and return 5 QQIDs
# note the latency incurred by retrieving the seed from the ANU server
rngQQID(5)

# return 10,000 QQIDs and transform them into UUIDs (takes less than two
# seconds); we assume that the RNG is in a sane state since we have just
previously initialized it with a true random number
x <- qq2uu(rngQQID(1e4, method = "n"))

## End(Not run)
```

---

sxtMap *sxtMap*

Description

sxtMap maps 6-character bit patterns to their corresponding Base64 characters (an "sextet"), or characters back to bit patterns.

Usage

```
sxtMap(x)
```

Arguments

x	(character or numeric) A vector of encoding characters, bit-patterns or indices. Note: indices are vector indices, not interpreted bit-patterns, i.e `sxtMap("000000") == sxtMap(1)`

Value

(character) A vector of Base 64 characters or their corresponding bit-patterns.

Description

If the input is a vector of 6-character bit patterns, sxtMap returns the corresponding character from the Base64 binary-to-text encoding. If the input is a vector of 1-character strings sxtMap returns the corresponding bit pattern. If the input is numeric, it is interpreted as indices into the sxt vector. The returned vector has the same length as the input but no checking for valid input is done.

Base64

Base64 is a binary to text encoding specified in RFC 1738; we use the URL and filename-safe alphabet version with the following encoding from `000000` to `111111`: `"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstu`
This is safe for most applications; one exception is Microsoft Excel where strings that begin with a `"-"` are silently convert to formulas. (QQIDs are not affected by this problem since they always begin with a lowercase alphabetic character.)

Author(s)

Examples

```
# sxtMap three bit patterns
sxtMap(c("101101", "011011", "010110"))   # "t"    "b"    "W"

# sxtMap three encoding characters
sxtMap(c("a", "b", "c"))  # "011010" "011011" "011100"

# print the entire r64 vector as one string
paste0(sxtMap(1:64), collapse = "")
```

`xlt2qq` *xlt2qq*

Description

`xlt2qq` converts a vector of 128-bit numbers (hexlets) in hexadecimal notation, UUID format, IPv6 addresses, or MD5 hashes to QQIDs.

Usage

```
xlt2qq(xlt)
```

Arguments

`xlt`	(character) a vector of UUIDs, MD5 hashes, IPv6 addresses, or generally 32 digit hexadecimal numbers

Value

(character) a vector of QQIDs

128-bit numbers and QQIDs

UUIDs, IPv6 addresses and MD5 hashes are specially formatted 128-bit numbers, referred to as hexlets. Randomly chosen 128-bit numbers have a collision probability that is small enough to make them useful as (practically) unique identifiers in applications where a centralized management of IDs is not feasible or not desirable. However since they are long strings of numerals and letters, without overt semantic content, they are hard to distinguish by eye. This creates difficulties when developing, or debugging with structured data, or for the curation of ID tagged information. The `qqid` package provides tools to convert the leading 20-bits of 128-bit numbers to two "Q-words", and the remainder to a string of 18 Base64 encoded characters. The "Q-words" - the letter Q evokes the word "cue" i.e. a hint or mnemonic - define a unique and invertible mapping to 2^10 integers (0, 1023). Thus two Q-words can encode 20 bits, or 5 hexadecimal letters:

```
.
.          [0-9a-f]    [0-9a-f]    [0-9a-f]    [0-9a-f]    [0-9a-f]
.  hex:  |--0x[1]--| |--0x[2]--| |--0x[3]--| |--0x[4]--| |--0x[5]--|
.  bit:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.        |----------int[1]-----------| |----------int[2]-----------|
.  int:            (0, 1023)                    (0,1023)
.    Q:      (aims, ..., zone)     .       (aims, ..., zone)     .   Base64...
.
```

Process

Input strings are first converted to plain hexadecimal strings. A leading "0x" is deleted, the "-" and ":" separators of UUIDs and IPv6 addresses respectively are deleted, and all letters are converted to lower case. It is an error if the result is not exactly a 32 digit hexadecimal `"[0-9a-f]\{32\}"` string. The first five hexadecimal letters are interpreted as two ten bit numbers, and mapped as indices into the 1024-element Q-Word vector. The QQID has two Q-words as a head representing digits 1:5 of the input, and the 18 Base64 encoded digits 6:32 of the input as its tail. Since the mapping is fully reversible, QQIDs have exactly the same statistical properties as the input. For details on QQID format see `is.QQID()`.

Input formats

A hexlet comprises 16 octets and is written in the hexadecimal numeral convention. A canonical MD5 hash is such a string of 32 hexadecimal characters. To improve readability, separators are inserted into UUIDs: `"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"` where `"x"` is a hexadecimal letter. A canonical expanded IPv6 address has the form: `"xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx"` where `"x"` is a hexadecimal letter. Conventions exist to omit leading zeros in IPv6 addresses, such shortened addresses are treated as an error. It is up to the user to expand them correctly before processing. There are many representations of hexadecimal numbers, most commonly they have a prefix of "0x". `xlt2qq()` converts all letters to lowercase on input.

Endianness

The `qqid` package uses its own functions to convert to and from bits, and is not affected by big-endian vs. little-endian processor architecture or variant byte order. All numbers are interpreted to have their lowest order digits on the right.

Author(s)

(c) 2019 Boris Steipe, licensed under MIT (see file LICENSE in this package).

See Also

qq2uu() to convert a vector of QQIDs to UUIDs.

Examples

```
# Convert three example UUIDs and one NA to the corresponding QQIDs
xlt2qq( c(xltIDexample(c(1, 3, 5)), NA) )

# A random hex-string is converted into a valid QQID
(x <- paste0(sample(c(0:9, letters[1:6]), 32, replace=TRUE), collapse=""))
(x <- xlt2qq(x))
is.QQID(x)                    # TRUE

# forward and back again
myID <- "0c460ed3-b015-adc2-ab4a-01e093364f1f"
myID == qq2uu(xlt2qq(myID))   # TRUE

# Confirm that the example hexlets are converted correctly
xlt2qq( xltIDexample(1:5) ) == QQIDexample(1:4)  # TRUE TRUE TRUE TRUE TRUE
```

---

xltIDexample *xltIDexample*

Description

xltIDexample returns synthetic, valid 128-bit numbers in hexadecimal notation (hexlets) in different common formats. The synthetic examples are easy to distinguish from "real" IDs to prevent their accidental use in an application.

Usage

```
xltIDexample(sel = 1:5)
```

Arguments

sel	(numeric, logical, or character) a subsetting vector

Details

The function stores five artificial sample IDs. Input is a subsetting vector that specifies which IDs to return. More than five IDs can be requested by applying the usual subsetting rules. The IDs can be converted to the exact same QQIDs provided by QQIDexample(). The formats available are "md5": 32 hex numerals; "hex": 32 hex numerals with "0x" prefix; "UUID": Universally Unique Identifier format; "IPv6": IPv6 formatted address; "hEx": 32 hex numerals with mixed case.

Value

(character) a named vector of formatted hexlets.

Author(s)

Boris Steipe (aut)

See Also

QQIDexample() Returns five QQIDs

Examples

```
xltIDexample()                                # the five stored hexlets
xltIDexample(2:3)                             # a hex number and a UUID
xltIDexample(c(TRUE, FALSE))                  # vector recycling
xltIDexample(sample(1:5, 17, replace = TRUE))   # seventeen in random order
xltIDexample("UUID") == qq2uu(QQIDexample(3))   # TRUE (correct conversion)
```

Index