uuidrev
Internet-Draft
Intended status: Informational
Expires: 12 October 2026

K. R. Davis
Cisco Systems

10 April 2026

Alternate UUID Encoding Methods
draft-davis-uuidrev-alt-uuid-encoding-methods-00

Abstract

This document presents considerations and best practices for alternate Universally Unique Identifier (UUID) encoding methods observed in the industry.

This document updates RFC9562 to provide suggested alternate encoding methods, best practices and the various implementation considerations required for choosing the correct encoding for an application.

When selected correctly, these alternate UUID encodings perform better on the wire, in a database, or within various other application logics versus the unnecessarily verbose text representation from RFC9562.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://example.com/LATEST. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-davis-uuidrev-alt-uuid-encoding-methods/.

Discussion of this document takes place on the Revise Universally Unique Identifier Definitions (uuidrev) Working Group mailing list (mailto:uuidrev@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/uuidrev/. Subscribe at https://www.ietf.org/mailman/listinfo/uuidrev/.

Source for this draft and an issue tracker can be found at https://github.com/uuid6/new-uuid-encoding-techniques-ietf-draft.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

This Internet-Draft will expire on 12 October 2026.

Copyright Notice

Table of Contents

1. Introduction

The original "hex-and-dash" (8-4-4-4-12) canonical format of UUIDs defined in [RFC9562] represents a 128-bit UUID value as a 288-bit text value. Although this format is useful for human readability, it is verbose for storage, transmission, and application parsing.

Because of this, developers have long used alternate UUID encodings. Many bespoke implementations—either within a single application or as one-off libraries—address shortcomings of the hex-and-dash form. However, most applications, databases, and standard libraries expose UUIDs exclusively using the formats described in IETF documents. If a feature is not specified by a well defined standard, implementers typically do not provide that feature and users do not see those capabilities. Consequently, although alternate encodings have been used and reimplemented for many years, there is no single applicable standard and implementations are inconsistent.

During the revision process that produced UUIDv6, UUIDv7, and UUIDv8 for [RFC9562], alternate encodings were a major topic of discussion. The level of interest showed that a separate document focused on alternate encodings was needed.

This document describes the most common alternate UUID encoding methods observed in the field and discusses the advantages and disadvantages of each. The latter half of the document presents best practices and considerations implementations should weigh when selecting, implementing, or defining new alternate UUID encodings not covered in Alternate UUID Encoding Methods. These best practices are based on years of research, community feedback, and inspection of over 55 alternate-encoding implementations across 17 alphabets (see Example UUID Base Encoding Tools, Libraries, and resources).

2. Terminology

2.1. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2.2. Abbreviations

The following abbreviations are used in this document:

UUID Universally Unique Identifier [RFC9562]

BaseXX	Base where XX references two numeric values with any leading 0 kept. For example Base02, Base10, Base16, Base32, Base64, etc.

3. Alternate UUID Encoding Methods

[RFC9562], Section 4 details that at its core, any given UUID is a 128 bit value which can be represented as Base02 (binary), Base10 (decimal/integer), Base16 (hex) and even a custom "hex-and-dash" string format; which is Base16 (hex) with a few extra dash characters added to create a unique UUID string format that is instantly recognizable. Further, the section goes on to discuss other string formats that often use the Hex and Dash format or Integer format.

While these are the "well-known" UUID formats, a UUID is fundamentally a 128-bit value and can be represented using many BaseXX alphabets.

The large number of possible BaseXX alphabets, together with the considerations described in Section 4, makes it impractical to cover every alphabet variant in a single document. Therefore, this document focuses on BaseXX alphabets that use US-ASCII ([RFC20]) and that are most commonly used in real-world implementations or were prototyped during the draft’s preparation.

UUID library implementers SHOULD consider adding support for at least one alternate encoding described here and MAY support additional encodings, including ones not listed. If you implement a BaseXX alphabet not covered in the sections below, consult the considerations in UUID Encoding Best Practices.

For a quick comparison of alphabets, see Table 1.

Encoding	Variant	Alphabet Order
Base16	[RFC9562], Section 4	0123456789ABCDEF with four dash/hyphen - characters added
Base16	[RFC4648], Section 8	0123456789ABCDEF
Base32	[RFC4648], Section 6	ABCDEFGHIJKLMNOPQRSTUVWXYZ234567
Base32	[RFC4648], Section 7	0123456789ABCDEFGHIJKLMNOPQRSTUV
Base32	[Base32human]	0123456789ABCDEFGHJKMNPQRSTVWXYZ
Base36	---	0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
Base52	---	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
Base58	[Base58btc]	123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
Base62	[Base62ieee]	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
Base62	[Base62sort]	0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
Base64	[RFC4648], Section 4	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
Base64	[RFC4648],	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_

	Section 5	
Base64	[Base64sort]	-0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz
Base85	[Z85]	0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ.-:+=^!/*?&<>()[]{}@%$#

Table 1: Alt UUID Encoding Alphabet Comparison

There also exist some algorithms that perform some pre-processing on the binary encoding of the UUID before encoding it as one or more of BaseXX Alphabets from the Table 1 for very specific application use-cases. The specifics about each algorithm are covered in their own documents but are included here for completeness and comparison purposes; see XML, HTML, CSS and Database Keys for the real-world problems these solve within databases and other applications.

The Table 2 table compares these different algorithms.

Encoding	Alphabet(s) and notes
[NCNAME]	[RFC4648], Section 6, [Base58btc], [RFC4648], Section 5 Version and Variant extracted and placed at the start and end of the layout to "Bookend". 26, 23 and 22 character outputs for the different alphabets in use.
[Base62id]	[Base62sort] 130-bit post-processed UUID value by appending 0b10, 22 characters after encoding

Table 2: Comparison of UUID Pre-processing Algorithms

Note: All UUID examples in this document are alternate encodings of [RFC9562], Section 4's Example UUID starting from either Figure 2 (binary) or Figure 3 (integer). Each section has a few example encodings; any UUID featured in RFC9562 can be found in the Appendix C table converted to the corresponding BaseXX alphabet encoding.

3.1. Base32

Base32 alphabets generally use 5 bits per character, producing a 26-character output when encoding a 128-bit UUID; padding may be present. These alphabets include standards-based Base32 ([RFC4648], Section 6), Base32hex ([RFC4648], Section 7), Base32 for Humans [Base32human] (formerly known as Douglas Crockford's Base32), Z-Base-32 [ZB32], and [GEOHASH].

Base32 alphabets vary the most in character choice because implementers can be selective about which characters are included. Character Exclusions and alphabet ordering (see Sorting and Ordering) are important considerations and differ across implementations.

Base32 is often case-insensitive: lowercase letters are frequently treated the same as uppercase because the alphabet does not require distinct upper- and lower-case characters. Special Characters are rarely used with Base32, except for Padding Characters or Checksum Characters features, which can often be omitted.

Base32 is a strong choice when available. The standard Base32 alphabet ([RFC4648], Section 6) is not recommended for new UUID implementations because its alphabet does not preserve binary sort order; libraries currently providing this encoding need not deprecate it, but new implementations should use Base32hex or Base32 for Humans instead. Base32hex ([RFC4648], Section 7) is recommended when sorting is required. Base32 for Humans ([Base32human]) is recommended when both sorting and human readability are required. See Figure 1 and Figure 2 for examples.

Z-Base-32 human-oriented base-32 encoding makes many changes to the underlying alphabet to accommodate a specific use case and is not recommended for UUIDs. Although [GEOHASH] is closer to Base32hex and excludes some characters like the Base32 for Humans alphabet, its removal of trailing zeros is an unusual behavior not used by other BaseXX alphabets and may make adoption difficult.

There are many other Base32 variants; consult UUID Encoding Best Practices when selecting an alphabet not listed here.

```
V0EKVBJTTG8T19R502GCI7JBUO
```

Figure 1: Example UUID encoded as Base32 Hex

```
Z0EMZBKXXG8X19V502GCJ7KBYR
```

Figure 2: Example UUID encoded as Base32 for Humans

3.2. Base36

Base36 alphabets are similar to Base32 but use about 5.1699 bits per character, reducing a 128-bit UUID to 25 characters. Base36 typically uses digits 09 followed by uppercase AZ, which affects case-sensitivity considerations (see Case Sensitivity). Alphabet ordering matters: using an alphabet ordered as AZ09 would change sort order and is therefore uncommon.

Base36 also compresses leading null bytes (see Section 4.9) which may be desirable to further reduce the size of the encoded UUID.

Base36 is reasonably available (Section 4.10) but does not have concrete specifications outlined by a standards body.

Base36 is recommended when variable-length UUIDs are not a problem for the application.

See Figure 3 for an example.

```
EOSWZOLG3BSX0ZN8OTQ1P8OOM
```

Figure 3: Example UUID encoded as Base36

3.3. Base52

Base52 uses the 52 alphabetic characters AZ and az, producing a 23-character UUID at about 5.700 bits per character.

Although not widely used, Base52 has useful properties: it contains no special characters (Special Characters) and excludes digits, avoiding issues with leading digits in some contexts. Its ordering can be chosen to preserve desirable sort behavior (see Sorting and Ordering).

Base52 also compresses leading null bytes (see Section 4.9) which may be desirable to further reduce the size of the encoded UUID.

Base52 does not have concrete specifications outlined by a standards body, but it is reasonably easy to implement.

Base52 is recommended when sorting is required, special characters and digits pose problems, and variable-length UUIDs are not a problem.

See Figure 4 for an example.

```
FraqvVvqUBoEOFXsPYOPwUm
```

Figure 4: Example UUID encoded as Base52

3.4. Base58

Base58, used by [Base58btc] (and [Flickr]), is a popular alphabet and similar to Base32 for Humans in its character exclusions. Base58 encodes a 128-bit UUID into a 22-character string (about 5.857 bits per character). Base58 typically omits Padding Characters, and most implementations follow the same alphabet, with the [Base58xrp] variant being a notable exception.

Base58 also compresses leading null bytes (see Section 4.9) which may be desirable to further reduce the size of the encoded UUID.

Base58 Encoding Availability varies, but the format is well-known and commonly used for UUIDs in practice.

Base58 is recommended when sorting and human readability are required, and variable-length UUIDs are not a problem.

See Figure 5 for an example.

B7wc88dU4e3NyJEj3e944DK

Figure 5: Example UUID encoded as Bitcoin’s Base58

3.5. Base62

Base62 is the largest BaseXX alphabet that contains no special characters (Special Characters). Base62 encodes a 128-bit UUID as a 22-character string (about 5.954 bits per character). During research the authors observed a number of libraries or discussions specifically using Base62 for UUIDs.

[Base62ieee] (defined by the IEEE) is one implementation and [Base62sort] is a variant that preserves sort order. Both variants use the same set of 62 characters but in a different order: [Base62ieee] orders the alphabet as AZaz09 while [Base62sort] orders it as 09AZaz to preserve lexicographic sort order (see Table 1). Additionally, [Base62ieee] includes padding while [Base62sort] does not.

[Base62ieee] also compresses leading null bytes (see Section 4.9) which may be desirable to further reduce the size of the encoded UUID.

[Base62ieee] is recommended when variable-length UUIDs are not a problem. [Base62sort] is recommended when sorting is required and special characters or digits pose problems.

See Figure 6 for an example.

HiLegqjbBwUc0Q8tJ3ffs6

Figure 6: Example UUID encoded as IEEE’s Base62

3.6. Base64

Base64 alphabets require special characters (Special Characters) to complete the 64-symbol set, so they typically do not use character exclusions. Base64 encodes a 128-bit UUID into a 22-character string (6 bits per character), similar in length to Base58 and Base62.

Because symbols are involved, many other permutations exist and this document cannot cover them all; see Usage when selecting an alphabet. The most common variants are the Base64 alphabet in [RFC4648], Section 4, Base64url in [RFC4648], Section 5 and Base64sort ([Base64sort]) (also known as Base64lex).

The standard Base64 alphabet ([RFC4648], Section 4) is not recommended for new UUID implementations; libraries currently providing this encoding need not deprecate it, but new implementations should use Base64url or Base64sort instead. Base64url ([RFC4648], Section 5) is recommended for most use cases because it is safe for URLs, filenames and Application Specific use cases. Base64sort ([Base64sort]) is recommended when lexicographical sorting that matches binary order is critical (for example, with UUIDv6 or UUIDv7).

Base64 alphabets enjoy widespread availability (Encoding Availability) and are far more common than Base32 as per an analysis of RFC4648 alphabet usage in the wild [RFC4648_Usage_Report].

See Figure 7 and Figure 8 for examples.

```
+B1Prn3sEdCnZQCgyR5r9g==
```

Figure 7: Example UUID encoded using the Base64 alphabet (with padding)

```
-B1Prn3sEdCnZQCgyR5r9g==
```

Figure 8: Example UUID encoded using the Base64 URL and Filename safe alphabet (with padding)

```
y0pEfbrg3S1bOF1VmGtfxV
```

Figure 9: Example UUID encoded using the Base64 Lexicographically sortable and Filename safe alphabet (without padding)

3.7. Base85

Base85 encodes 4 bytes as 5 characters (about 6.409 bits per character), so a 128-bit UUID can be represented in 20 characters. These alphabets include many special characters (Special Characters), and provide the most compact textual representation among the alphabets discussed. Common variants include Z85, [RFC1924] Base85, and ASCII85.

Some Base85 variants include compression techniques for certain byte sequences (see Compressing Characters). Those techniques are not universal across all Base85 alphabets.

Z85 is the recommended Base85 variant for UUIDs when symbols are not a concern.

See Figure 10 for an example.

```
{-iekEE4M)R!2>3:Stl>
```

Figure 10: Example UUID encoded as Base85 using Z85 alphabet

4. UUID Encoding Best Practices

There are several factors to consider when choosing an encoding for a UUID. Requirements that matter for one application may be irrelevant for another; the authors have grouped the most important considerations into best-practice categories in the sections that follow.

Each section focuses on UUID-specific concerns, but many of the topics generalize to selecting alternate encodings for other data.

4.1. Usage

Understanding how a UUID will be used is the most important step when choosing an alternate encoding. That analysis affects many of the considerations that follow.

For example, a UUID used as a logging prefix should be compact to reduce log size, while a UUID that must be read or spoken by humans should avoid characters that are easily confused. Shorter encodings with character exclusions are preferable when values are recited by phone or written by hand to reduce transcription errors.

Datastores (see [RFC9562]) care about case sensitivity, alphabet ordering, and encoded length because these factors affect sorting, storage, and performance at scale. UUIDs transmitted between machines benefit from extremely compact encodings to reduce bytes on the wire and improve behavior on limited-MTU or high-latency networks.

For resource-constrained devices such as IoT endpoints, prefer encodings that are computationally inexpensive to encode and decode.

For Large Language Model (LLM) applications, UUID encoding directly affects token consumption because each token maps to a variable number of characters and special characters often consume disproportionately more tokens. Shorter encodings that avoid special characters generally consume fewer tokens and reduce inference costs; .

For a given application use case (database key, network transmission, logging, etc.), implementations SHOULD choose exactly one BaseXX encoding and MUST NOT mix different BaseXX encodings for the same purpose. Mixing encodings breaks assumptions about encoded length

and sort order and complicates comparison logic. For example, if an application stores UUIDs as Base32 in a database, it should not also accept or produce Base62-encoded UUIDs for the same field.

The general statement by the authors is as follows: there is no single correct choice to cover every scenario.

4.2. Maximum Character Length

The number of characters/symbols in a given encoding alphabet is directly correlated to the size of the alternate UUID encoding output.

As a starting point the encodings defined by RFC9562 are Binary, Integer, and Hex. Base02 (Binary) version of a UUID is 128 characters in length consisting of 0s and 1s. The Base10 (decimal/integer) version of a UUID is 39 characters consisting of characters 0 through 9 while the Base16 (Hex) version of a UUID shortens this to 32 characters by using characters 0 through 9 and A through F.

Base32 libraries tend to produce a UUID output of 26 characters in length and base64 drops the output UUID to 22 characters.

With this trend one may be tempted to simply pick the highest BaseXX encoding available and get the smallest encoding output possible however as the output encoding decreases, the base alphabet increases. At a specific point both numeric values, upper and lowercase values may be present and any number of unique symbols can be present in the base alphabet. The addition and ordering of these can have rippling effects that one must properly consider. Further, some of these alphabets require unique math to properly apply to the fixed-length 128 bit UUID which may increase computational complexity in unfavorable ways along with padding to output a similarly fixed-length value.

Simply picking the biggest base alphabet to get the smallest encoding only works if the only consideration an implementation cares about is the actual size of the output UUID.

To glean the maximum size of a given BaseXX encoding with a 128-bit UUID, convert the binary form of UUID MAX from [RFC9562], Section 5.10.

Table 3 details the maximum length of various BaseXX Encoding methods with UUID MAX, without padding present.

Encoding	Variant	Maximum Character Length
Base16	[RFC9562], Section 4	36
Base16	[RFC4648], Section 8	32
Base32	[RFC4648], Section 6	26
Base32	[RFC4648], Section 7	26
Base32	[Base32human]	26
Base36	---	25
Base52	---	23
Base58	[Base58btc]	22
Base62	[Base62ieee]	22
Base62	[Base62sort]	22
Base64	[RFC4648], Section 4	22
Base64	[RFC4648], Section 5	22
Base64	[Base64sort]	22
Base85	[Z85]	20

Table 3: Alt UUID Encoding Length Comparison

4.3. Padding Characters

Padding characters go hand in hand with the previous section involving the maximum character length of an output alternate UUID encoding.

The most common padding character is the equal sign (=) however it could theoretically be any character possible. It is recommended to use the equal sign since it is the most well-known padding character among the various BaseXX encodings.

Further, this character is almost always in the least significant, right-most section of a given alternate UUID encoding. The padding SHOULD stay in this position and moving the character can have ramifications for sorting.

For example, Base64sort ([Base64sort]) allows for either the equal sign (=) or tilde (~) as a padding character, but only the tilde is valid for sorting because the equal sign is lexicographically less than all other characters in the alphabet.

Since UUIDs defined by [RFC9562] are always 128 bits, padding MAY be omitted and the resulting output will always be the same length.

Table 4 compares padding usage among the BaseXX encoding methods in this document.

Encoding	Variant	Padding
Base16	[RFC9562], Section 4	N
Base16	[RFC4648], Section 8	N
Base32	[RFC4648], Section 6	Y (=)
Base32	[RFC4648], Section 7	Y (=)
Base32	[Base32human]	N
Base36	---	N
Base52	---	N
Base58	[Base58btc]	N
Base62	[Base62ieee]	Y
Base62	[Base62sort]	N
Base64	[RFC4648], Section 4	Y (=)
Base64	[RFC4648], Section 5	Y (=)
Base64	[Base64sort]	Y (= or ~)
Base85	[Z85]	N

Table 4: Alt UUID Encoding Padding Comparison

4.4. Checksum Characters

Some newer BaseXX encodings include checksum characters that are used to ensure the input/output encoding and decoding is working as expected.

The actual character and algorithm used vary among BaseXX implementations.

Including checksums increases both the maximum size of the output encoding and the computation requirements for encoding/decoding alternate UUID formats.

For alternate UUIDs transmitted among two machines it may be beneficial to include a checksum character to validate the given UUID has not been modified during transport before using it for various operations thus increasing the resiliency of the alternate UUID generation and parsing process.

For applications that simply care about generating UUIDs but do not need to parse, checksums MAY be omitted entirely.

Table 5 compares checksum usage among the BaseXX encoding methods in this document.

Encoding	Variant	Checksums
Base16	[RFC9562], Section 4	N
Base16	[RFC4648], Section 8	N
Base32	[RFC4648], Section 6	N
Base32	[RFC4648], Section 7	N
Base32	[Base32human]	Y (*,~,$,=,U,u)
Base36	---	N
Base52	---	N
Base58	[Base58btc]	Y (SHA256-based)
Base62	[Base62ieee]	N
Base62	[Base62sort]	N
Base64	[RFC4648], Section 4	N
Base64	[RFC4648], Section 5	N
Base64	[Base64sort]	N
Base85	[Z85]	N

Table 5: Alt UUID Encoding Checksum Comparison

4.5. Special Characters

After all of the alpha-numeric characters are used a BaseXX alphabet must resort to using special characters such as but not limited to underscore (_), hyphen (-), plus (+), forward slash (/), dollar sign ($), etc. This usually occurs around 62 characters (ten digits 0-9, 26 lowercase English characters, 26 uppercase English characters) but may happen with earlier BaseXX alphabets when specific characters are excluded.

The inclusion of these characters decrease the overall size of the encoded UUID but have implications in regards to not just sorting and readability but often application usage. For example Base64 as defined by [RFC4648] has two alphabets which change the special characters to ensure safe usage with URLs and Filenames.

Another example is the ability to double-click a UUID and copy the value. When special characters are present the text highlight starts and stops between these characters. In such a scenario, if this is important one should implement a BaseXX Alphabet with no such characters.

When evaluating a baseXX encoding, care must be taken to ensure the special characters are compatible with the desired use case.

Table 6 compares special character inclusions among the BaseXX encoding methods in this document.

Encoding	Variant	Special Characters
Base16	[RFC9562], Section 4	Y (dash)
Base16	[RFC4648], Section 8	N
Base32	[RFC4648], Section 6	N
Base32	[RFC4648], Section 7	N
Base32	[Base32human]	N
Base36	---	N
Base52	---	N
Base58	[Base58btc]	N
Base62	[Base62ieee]	N
Base62	[Base62sort]	N
Base64	[RFC4648], Section 4	Y (plus, forward-slash)
Base64	[RFC4648], Section 5	Y (dash, underscore)
Base64	[Base64sort]	Y (dash, underscore)
Base85	[Z85]	Y (numerous, see Table 1)

Table 6: Alt UUID Encoding Special Character Comparison

4.6. Character Exclusions

Newer BaseXX alphabets may exclude characters, retaining only one character from each group of visually similar characters.

This often occurs when multiple characters are similar in shape, which may lead to issues when humans are required to read the characters aloud or manually transpose the characters by hand. Some characters may be excluded for other non-obvious reasons like avoiding potential vulgar words or other application-specific issues.

For example it may be advantageous to remove 0 as a possible option to eliminate the possibility of leading 0s in the output data which may lead to problems in specific use cases where leading 0s are not permitted or may be inadvertently removed by the application.

Finally, in some smaller BaseXX alphabets it is possible to exclude characters because they are not required to fill out the space. The notable example is Base32hex defined by [RFC4648], Section 7 which only uses A through V because the remaining slots are filled by numeric 0-9.

The following character groupings illustrate some of the problematic characters that are similarly shaped.

- 0, O, o
- 1, I, i, L, l
- 2, Z, z
- 5, S, s
- V, v, U, u

Unfortunately there is no consistent method for which character is omitted or removed. Some BaseXX alphabets may remove the numeric characters while others may remove one or both of the English characters, often replacing them with symbols in larger BaseXX alphabets to fill the gaps.

Implementations should vet BaseXX alphabets with character exclusions thoroughly to ensure any possible inclusion of symbols does not cause problems while also verifying that sorting is not impacted if sorting is a requirement.

Table 7 compares character exclusions among the BaseXX encoding methods in this document.

Encoding	Variant	Character Exclusions
Base16	[RFC9562], Section 4	N
Base16	[RFC4648], Section 8	N
Base32	[RFC4648], Section 6	N
Base32	[RFC4648], Section 7	N
Base32	[Base32human]	Y (Ii,Ll,Oo,Uu)
Base36	---	N
Base52	---	N
Base58	[Base58btc]	Y (0,I,O,l)
Base62	[Base62ieee]	N
Base62	[Base62sort]	N
Base64	[RFC4648], Section 4	N
Base64	[RFC4648], Section 5	N
Base64	[Base64sort]	N
Base85	[Z85]	N

Table 7: Alt UUID Encoding Character Exclusion Comparison

4.7. Case Sensitivity

Case sensitivity becomes important when the alphabet includes more than 32 characters. For alphabets at or below Base32 (for example, Base16), uppercase and lowercase characters are often treated interchangeably; for larger alphabets, uppercase and lowercase letters represent distinct values.

In most scenarios, uppercase letters are ordered before lowercase letters. This ordering directly impacts sorting because many applications sort uppercase characters before lowercase characters. Sorting is further covered in Sorting and Ordering.

The BaseXX alphabet used also changes comparison logic: "aBc" does not necessarily equal "AbC" in all encodings.

Table 8 compares case sensitivity among the BaseXX encoding methods in this document.

Encoding	Variant	Case Sensitive
Base16	[RFC9562], Section 4	N
Base16	[RFC4648], Section 8	N
Base32	[RFC4648], Section 6	N
Base32	[RFC4648], Section 7	N
Base32	[Base32human]	N
Base36	---	N
Base52	---	Y
Base58	[Base58btc]	Y
Base62	[Base62ieee]	Y
Base62	[Base62sort]	Y
Base64	[RFC4648], Section 4	Y
Base64	[RFC4648], Section 5	Y
Base64	[Base64sort]	Y
Base85	[Z85]	Y

Table 8: Alt UUID Encoding Case Sensitivity Comparison

4.8. Sorting and Ordering

Lexicographical sorting of UUIDs is a very important factor to many applications and the ordering of the BaseXX alphabet directly impacts the sorting of the encoded UUID output such as those created by UUIDv6 and UUIDv7 where lexicographically sortable UUIDs are a key feature of the underlying algorithm.

The BaseXX alphabets generally consist of two or more of the following 4 components with optional omissions as defined by Section 4.6:

1. Numeric Characters: 0-9
2. Uppercase Character: A-Z
3. Lowercase Characters: a-z
4. Symbols: underscore (_), hyphen (-), plus (+), forward slash (/), dollar sign ($), etc.

The ordering of these is traditionally Numeric, Uppercase, Lowercase and Symbols. However for ordering purposes, some symbols may be placed at a different position to adhere to their position found in US-ASCII ([RFC20]).

A common sorting example is a symbol like the dash (-) character may sort before numbers while other special characters such as an underscore (_) will be sorted after the uppercase characters but before the lowercase characters if sorted via their ASCII values.

If an implementation would like to ensure the sort order of the encoded value remains the same as the Base02 (binary) or Base10 (decimal) value one should scrutinize the position of the underlying BaseXX alphabets, their ordering and the included symbols.

Figure 11 illustrate the ordering trends for some common BaseXX Alphabets that adhere to the US-ASCII character set and their ordering of the various components.

```
- Base10:      0123456789
- Base32hex:   0123456789ABCDEFGHIJKLMNOPQRSTUV
- Base36:      0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
- Base62sort:  0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
- Base64sort: -0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz
```

Figure 11: Alphabet Ordering Comparison

Table 9 compares binary sorting among the BaseXX encoding methods in this document.

Encoding	Variant	Sorts the Same as Binary
Base16	[RFC9562], Section 4	Y
Base16	[RFC4648], Section 8	Y
Base32	[RFC4648], Section 6	N
Base32	[RFC4648], Section 7	Y
Base32	[Base32human]	Y
Base36	---	Y
Base52	---	Y
Base58	[Base58btc]	Y
Base62	[Base62ieee]	N
Base62	[Base62sort]	Y
Base64	[RFC4648], Section 4	N
Base64	[RFC4648], Section 5	N
Base64	[Base64sort]	Y
Base85	[Z85]	N

Table 9: Alt UUID Encoding Sorting Comparison

4.9. Compressing Characters

Some BaseXX alphabets may use varying techniques for compressing null bytes (0x00), whitespace, or long runs of the continuous characters which can sometimes yield values smaller than what is cited in Section 4.2.

Base36, Base52, Base58, and A secure, lossless, and compressed Base62 encoding will trim leading zero characters before attempting to encode the data.

Base85 alphabets may employ a technique to compress a continuous four byte sequence of ASCII Space characters as the character lowercase y.

Base85 alphabets may also compress a continuous four byte sequence of ASCII Zero characters as the character lowercase z.

The leading zero compression technique can be observed in Appendix C.2 using NIL UUID input to illustrate the output featuring a much smaller value than the outputs found in other test vectors.

This variability may be desirable to produce even smaller UUIDs however this could also prove a problem if an application expects a UUID of a specific fixed-length value found in Section 4.2.

4.10. Encoding Availability

The BaseXX alphabet encoding availability may be the second greatest hurdle implementations navigate as they chose a viable alternate encoding for UUID.

For example, traditional Base64url safe sees far more standard implementations than Base32hex as per [RFC4648_Usage_Report] while even fewer have implemented Base62 or Base36 variants.

The ability to utilize a given encoding or provide a customized alphabet may be a limiting factor in implementations. In this scenario implementors will need to write their own, leverage third-party libraries or select a viable alternative.

This is being addressed in some ways by standardizing documents such as [Base32human] or [Base64sort] but there are still many BaseXX alphabets that do not have concrete specifications outlined by a standards body. Further, it takes time for a languages and applications to implement and adopt these standards and for the standards to be widely adopted by the community.

4.11. Computation

The various base encodings have more or less computational requirements based on the size of the alphabet, the total number of bits encoded per character, if checksums are involved, and if floating point math is required (e.g radix-xx computations).

For reference, Base02 uses 1 bit per character, Base10 uses about 3.322 bits per character, Base16 uses 4 bits per character, Base32 uses 5 bits, and Base64 uses 6 bits. While nonpower-of-two alphabets like Base36, Base58, or Base62 use approximately 5.169, 5.857, and 5.954 bits per character respectively, the computation is log_2(XX) where XX is the alphabet length.

Table 10 compares bits per character among the BaseXX encoding methods in this document.

Encoding	Variant	Bits per Character
Base16	[RFC9562], Section 4	4.000
Base16	[RFC4648], Section 8	4.000
Base32	[RFC4648], Section 6	5.000
Base32	[RFC4648], Section 7	5.000
Base32	[Base32human]	5.000
Base36	---	5.169
Base52	---	5.700
Base58	[Base58btc]	5.857
Base62	[Base62ieee]	5.954
Base62	[Base62sort]	5.954
Base64	[RFC4648], Section 4	6.000
Base64	[RFC4648], Section 5	6.000
Base64	[Base64sort]	6.000
Base85	[Z85]	6.409

Table 10: Alt UUID Encoding Bits Comparison

4.12. Parsing

This document focuses on generating UUIDs via alternate formats rather than parsing UUIDs of alternate encoding formats. The generic default for any existing uuid_parse(uuid) function SHOULD remain that of [RFC9562], Section 4 hex-and-dash string format.

Implementors MAY provide parse functionality to parse UUIDs of alternate formats such as uuid_parse(uuid, encoding="base64url"). The distribution (and naming) of the encoding algorithm among disparate systems is outside of the scope of this document.

In practice this is seldom an issue because most implementations that need to parse UUIDs are aware of the encoding format used by their peers. For example, the sender and receiver of a Base64url-encoded UUID are often within the same application boundary. It is uncommon for two different systems to exchange UUIDs in a way that requires the receiver to parse an alternate-format UUID back into binary; typically the receiver treats the received UUID as an opaque string.

4.13. Application Specific

Some applications have very specific restrictions which may influence the alternate UUID encoding selection. This section features a limited list compiling some known restrictions that implementations may consider while working with alternate UUID encodings.

4.13.1. LLM Token Efficiency

Large Language Models (LLMs) process text using tokens: sub-word or character-level units whose boundaries depend on the model's tokenizer. Because UUID alternate encodings mix digits, uppercase letters, lowercase letters, and sometimes special characters in patterns that rarely appear in natural language, tokenizers often split them into many small tokens. Greater token consumption translates directly to higher inference cost, increased latency, and reduced effective context-window utilization.

Tokenizer behavior varies substantially across vendors and model families. The token counts below were measured using the following tokenizers, which were current when this document was prepared:

* Anthropic (https://platform.claude.com/docs/en/build-with-claude/token-counting): Claude OPUS 4.6
* OpenAI (https://platform.openai.com/tokenizer): GPT5.x & O1/3
* xAI (https://docs.x.ai/developers/rest-api-reference/inference/other#tokenize-text): grok-4.20-0309-reasoning
* Google (https://ai.google.dev/api/tokens): Gemini 3.1 Pro Preview
* Generic: ceil(character_length / 4), the common "1 token ~= 4 characters" heuristic

Note: The token counts in Table 11 use the MAX UUID from Appendix C.3 which consists largely of repeating characters. Repeating characters compress very aggressively in Byte-Pair Encoding based tokenizers, so these values represent a best-case (minimum token) scenario. Table 12 uses the Example UUID from Appendix C.1 whose mixed characters are more representative of typical UUIDs. Real-world token counts will more closely resemble the latter.

Table 11 details the token count for the MAX UUID across various BaseXX Encoding methods and tokenizers.

Encoding	Variant	Anthropic	OpenAI	xAI	Google	Generic
Base16	[RFC9562], Section 4	21	10	10	13	9
Base16	[RFC4648], Section 8	7	4	2	9	8
Base32	[RFC4648], Section 6	10	9	9	27	7
Base32	[RFC4648], Section 7	27	13	13	14	7
Base32	[Base32human]	15	13	13	14	7
Base36	---	10	10	10	27	7
Base52	---	21	15	15	16	6
Base58	[Base58btc]	18	14	14	16	6
Base62	[Base62ieee]	20	15	12	13	6
Base62	[Base62sort]	22	17	17	19	6
Base64	[RFC4648], Section 4	4	4	4	4	6
Base64	[RFC4648], Section 5	4	3	3	4	6
Base64	[Base64sort]	12	11	6	12	6
Base85	[Z85]	20	20	20	21	5

Table 11: Alt UUID Encoding Token Count: MAX UUID

Table 12 details the token count for a typical UUID (Appendix C.1) across various BaseXX Encoding methods and tokenizers.

Encoding	Variant	Anthropic	OpenAI	xAI	Google	Generic
Base16	[RFC9562], Section 4	26	25	21	33	9
Base16	[RFC4648], Section 8	22	22	15	29	8
Base32	[RFC4648], Section 6	22	17	16	19	7
Base32	[RFC4648], Section 7	20	18	17	22	7
Base32	[Base32human]	21	18	16	22	7
Base36	---	22	17	17	18	7
Base52	---	20	14	15	14	6
Base58	[Base58btc]	17	15	14	18	6
Base62	[Base62ieee]	20	14	16	17	6
Base62	[Base62sort]	21	16	15	16	6
Base64	[RFC4648], Section 4	20	16	14	18	6
Base64	[RFC4648], Section 5	20	16	14	18	6
Base64	[Base64sort]	21	16	14	17	6
Base85	[Z85]	19	15	14	16	5

Table 12: Alt UUID Encoding Token Count: Example UUID

Key takeaways:

* The commonly cited "1 token ~= 4 characters" heuristic severely underestimates real token costs for UUIDs. The generic column predicts 5-9 tokens, but official tokenizers return 14-33 tokens for a typical UUID, a 2-4x underestimate.

* Higher-base encodings (Base52+) generally produce fewer tokens than lower-base encodings (Base16, Base32) for typical UUIDs because the shorter character length outweighs the added alphabet complexity. For example, Base16 with hyphens costs 21-33 tokens across vendors while Base58-Base85 range from 14-19.
* Removing the hyphens from the standard Base16 UUID representation saves 4-7 tokens depending on the vendor (e.g., xAI: 21 to 15, Google: 33 to 29 for the example UUID).
* Base64 standard variants ([RFC4648] Section 4 and 5) are exceptionally efficient for repetitive inputs (3-4 tokens for MAX UUID) but converge with other high-base encodings (14-20 tokens) for typical UUIDs.
* Token counts vary substantially across vendors for the same input; for example, the standard Base16 UUID representation ranges from 21 tokens (xAI) to 33 tokens (Google) for the same UUID. Applications sensitive to token costs should measure with their target model’s tokenizer.

4.13.2. URI, URL, URN

Uniform Resource Identifiers (URIs) have a specific list of reserved characters defined by [RFC3986], Section 2.2 which require percent encoding to be used.

It is advisable to avoid BaseXX alphabets that include these symbols.

4.13.3. DNS Record

Domain Name Systems (DNS) are case insensitive with the period or dot character (.) being a reserved symbol. [RFC9499], Section 2

Thus BaseXX alphabets that utilize this character or use upper and lowercase values as defined by Case Sensitivity should be avoided.

4.13.4. XML, HTML, CSS

Extensible Markup Language ([XML]) namespaces cannot start with a leading numeric character.

While older versions of Hypertext Markup Language ([HTML]) had restrictions on element identifiers starting with a leading digit, HTML5 does not have any such restrictions. The same goes for Cascading Style Sheet ([CSS]) selectors identifiers where older versions would have issues when identifiers start with digits.

Thus for XML, HTML, and CSS it is advantageous to use some advanced methods such as:

* Leveraging the alternate UUID encoding technique defined via [NCNAME] (UUID-NCName-32, UUID-NCName-58 or UUID-NCName-64) to ensure that the first digit is always an uppercase alphabet character.
* An alternative algorithm to the NCNAME method is to prefix a value like b10 as seen in [Base62id] to ensure the first character is always a letter.
* Utilize Section 3.3 which ensures the output UUID does not start with a special character or digit.
* Simply prefix the first non-digit/non-special character in the given alphabet. For example, if uppercase A is available, prefix this character to satisfy the constraints.

4.13.5. Database Keys

Databases often have specific requirements for keys such as case sensitivity, sorting, and maximum length.

When using alternate UUID encodings as database keys, it is important to consider these requirements and select an encoding that meets them.

Database implementors should consider [Base62id] when using alternate UUID encodings as database keys to ensure the first character is an uppercase alphabet character and the output is compact.

5. Security Considerations

Section Checksum Characters addresses the primary security-related concern in this document: data-integrity validation for UUIDs transmitted over the wire.

No additional security considerations are identified.

6. IANA Considerations

This document has no IANA actions.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC9562] Davis, K., Peabody, B., and P. Leach, "Universally Unique IDentifiers (UUIDs)", RFC 9562, DOI 10.17487/RFC9562, May 2024, <https://www.rfc-editor.org/rfc/rfc9562>.

7.2. Informative References

[Base32human]
Crockford, D. and K. Davis, "Base32 for Humans", April 2026, <https://datatracker.ietf.org/doc/draft-crockford-davis-base32-for-humans/>.

[Base58btc]
Bitcoin, "Bitcoin Base58 Implementation", commit fae71d3, November 2008, <https://github.com/bitcoin/bitcoin/blob/master/src/base58.cpp>.

[Base58xrp]
XRP Ledger, "base58 Encodings", 2024, <https://xrpl.org/docs/references/protocol/data-types/base58-encodings>.

[Base62id] Prokhorenko, S., "Base62id Encoding Specification for Binary UUIDs", April 2026, <https://github.com/sergeyprokhorenko/Base62id/>.

[Base62ieee]
IEEE, "A secure, lossless, and compressed Base62 encoding", November 2008, <https://ieeexplore.ieee.org/document/4737287>.

[Base62sort]
Wu, P.-C., "A base62 transformation format of ISO 10646 for multilingual identifiers", August 2001, <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.408>.

[Base64sort]
Davis, K., "A Sortable Base64 Alphabet", December 2025, <https://datatracker.ietf.org/doc/draft-brown-davis-base64-sort/>.

[CSS] W3C, "Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification", June 2011, <https://www.w3.org/TR/CSS2/syndata.html#value-def-identifier>.

[Flickr] Kellen, "manufacturing flic.kr style photo URLs", n.d., <https://www.flickr.com/groups/api/discuss/72157616713786392/>.

[GEOHASH] Geohash, "Geohash", commit 3f0ce0b, November 2009, <https://github.com/vinsci/geohash/blob/master/Geohash/geohash.py>.

[HTML] whatwg, "HTML Living Standard", November 2025, <https://html.spec.whatwg.org/>.

[NCNAME] Taylor, D., "Compact UUIDs for Constrained Grammars", September 2025, <https://datatracker.ietf.org/doc/draft-taylor-uuid-ncname/>.

[RFC1924] Elz, R., "A Compact Representation of IPv6 Addresses", RFC 1924, DOI 10.17487/RFC1924, April 1996, <https://www.rfc-editor.org/rfc/rfc1924>.

[RFC20] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/rfc/rfc20>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[RFC4648_Usage_Report] "RFC4648 Alphabet Usage Report", commit de0a2760, November 2025, <https://gitlab.com/julian.reschke/base-encodings-terminology/-/blob/main/classifcation.md?ref_type=heads>.

[RFC9499] Hoffman, P. and K. Fujiwara, "DNS Terminology", BCP 219, RFC 9499, DOI 10.17487/RFC9499, March 2024, <https://www.rfc-editor.org/rfc/rfc9499>.

[XML] W3C, "Namespaces in XML 1.0 (Third Edition)", December 2009, <https://www.w3.org/TR/2009/REC-xml-names-20091208/>.

[Z85] iMatix Corporation, "32/Z85", 2013, <https://rfc.zeromq.org/spec/32/>.

[ZB32] O'Whielacronx, Z., "human-oriented base-32 encoding", November 2009, <https://philzimmermann.com/docs/human-oriented-base-32-encoding.txt>.

Appendix A. Acknowledgments

The authors gratefully acknowledge the contributions of

* Yulian Kuncheff for their work on the UUID Formatter test tool (https://uuidformattester.yuli.dev/) which was valuable in comparing various UUID encoding formats side-by-side.
* LiosK for early prototyping of various BaseXX Alphabets
* Sergey Prokhorenko for continuously ensuring we work on this document.

As well as all of those in the IETF community and on GitHub who contributed to the discussions which resulted in this document.

Appendix B. Changelog

draft-00:

* Initial Release

Appendix C. Test Vectors

The test vectors in the upcoming sections map the UUIDs found in RFC9562 Sections to their appropriate BaseXX Alphabet Encoding without padding.

All of the BaseXX Alphabets described in this document and others not described can be viewed at the online tool the Authors are using for prototyping and comparison: https://uuidformattester.yuli.dev/ (Source Code: https://github.com/daegalus/uuid-format-tester)

The NCNAME test vectors (UUID-NCName-32, UUID-NCName-58, and UUID-NCName-64) are direct copies from the appendix of [NCNAME].

Each section also includes test vectors for [Base62id] which are not included in the main body of the document but are included here for reference.

C.1. Generic UUID

Encoding	Variant	RFC9562, Section 4
Base16	[RFC9562], Section 4	f81d4fae-7dec-11d0-a765-00a0c91e6bf6
Base16	[RFC4648], Section 8	f81d4fae7dec11d0a76500a0c91e6bf6
Base32	[RFC4648], Section 6	7AOU7LT55QI5BJ3FACQMSHTL6Y
Base32	[RFC4648], Section 7	V0EKVBJTTG8T19R502GCI7JBUO
Base32	[Base32human]	Z0EMZBKXXG8X19V502GCJ7KBYR
Base36	---	EOSWZOLG3BSX0ZN8OTQ1P8OOM
Base52	---	FraqvVvqUBoEOFXsPYOPwUm
Base58	[Base58btc]	Xe22UfxT3rxcKJEAfL5373
Base62	[Base62ieee]	HiLegqjbBwUc0Q8tJ3ffs6
Base62	[Base62sort]	7YBUWgZR1mKSqGyj9tVViw
Base64	[RFC4648], Section 4	+B1Prn3sEdCnZQCgyR5r9g
Base64	[RFC4648], Section 5	-B1Prn3sEdCnZQCgyR5r9g
Base64	[Base64sort]	y0pEfbrg3S1bOF1VmGtfxV
Base85	[Z85]	{-iekEE4M)R!2>3:Stl>

Table 13: UUID Test Vectors for RFC9562, Section 4

Encoding	Output
UUID-NCName-32	b7aou7lt55qoqoziauder427wk
UUID-NCName-58	B7wc88dU4e3NyJEj3e944DK
UUID-NCName-64	B-B1Prn3sHQdlAKDJHmv2K
[Base62id]	N8JYxDHbz7rx9rGIw80uxC

Table 14: UUID-NCName, Base62id Test Vectors for RFC9562, Section 4

C.2. NIL UUID

Encoding	Variant	RFC9562, Section 5.9
Base16	[RFC9562], Section 4	00000000-0000-0000-0000-000000000000
Base16	[RFC4648], Section 8	00000000000000000000000000000000
Base32	[RFC4648], Section 6	AAAAAAAAAAAAAAAAAAAAAAAA
Base32	[RFC4648], Section 7	00000000000000000000000000
Base32	[Base32human]	00000000000000000000000000
Base36	---	0
Base52	---	A
Base58	[Base58btc]	1
Base62	[Base62ieee]	0
Base62	[Base62sort]	0
Base64	[RFC4648], Section 4	AAAAAAAAAAAAAAAAAAAAAA
Base64	[RFC4648],	AAAAAAAAAAAAAAAAAAAAAA

	Section 5	
Base64	[Base64sort]	----------
Base85	[Z85]	00000000000000000000

Table 15: UUID Test Vectors for RFC9562, Section 5.9: Nil UUID

Encoding	Output
UUID-NCName-32	aaaaaaaaaaaaaaaaaaaaaaaaa
UUID-NCName-58	A111111111111111_____A
UUID-NCName-64	AAAAAAAAAAAAAAAAAAAAA
[Base62id]	Fa84QWiAxLXUJaHZmEVPEG

Table 16: UUID-NCName, Base62id Test Vectors for RFC9562, Section 5.9: Nil UUID

C.3. MAX UUID

Encoding	Variant	RFC9562, Section 5.10
Base16	[RFC9562], Section 4	FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF
Base16	[RFC4648], Section 8	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Base32	[RFC4648], Section 6	77777777777777777777777774
Base32	[RFC4648], Section 7	VVVVVVVVVVVVVVVVVVVVVVVVVS
Base32	[Base32human]	ZZZZZZZZZZZZZZZZZZZZZZZZZW
Base36	---	p777777777777777777777p
Base52	---	GBIWTpZqojFGQPQPXtvbJAv
Base58	[Base58btc]	P8AQGAut7N92awznwCnjuQP

Base62	[Base62ieee]	HxECNQWFdpvuJxIw3HPrmH
Base62	[Base62sort]	7n42DGM5Tflk9n8mt7Fhc7
Base64	[RFC4648], Section 4	////////////////////w
Base64	[RFC4648], Section 5	_____w
Base64	[Base64sort]	zzzzzzzzzzzzzzzzzzzzk
Base85	[Z85]	s8W-!s8W-!s8W-!s8W-!

Table 17: UUID Test Vectors for RFC9562, Section 5.10: MAX UUID

Encoding	Output
UUID-NCName-32	p777777777777777777777777p
UUID-NCName-58	P8AQGAut7N92awznwCnjuQP
UUID-NCName-64	P_____P
[Base62id]	NNC6dn4GR1JETNQMfLl6qN

Table 18: UUID-NCName, Base62id Test Vectors for RFC9562, Section 5.10: MAX UUID

C.4. UUIDv1

Encoding	Variant	RFC9562, Section A.1
Base16	[RFC9562], Section 4	C232AB00-9414-11EC-B3C8-9F6BDECED846
Base16	[RFC4648], Section 8	C232AB00941411ECB3C89F6BDECED846
Base32	[RFC4648], Section 6	YIZKWAEUCQI6ZM6IT5V55TWYIY
Base32	[RFC4648],	O8PAM04K2G8UPCU8JTLTTJMO8O

	Section 7	
Base32	[Base32human]	R8SAP04M2G8YSCY8KXNXXKPR8R
Base36	---	BHW3F9QSZLQPYH8GTZ35HZ4UE
Base52	---	EddHArfCMSZGGUJvWdXrHHq
Base58	[Base58btc]	Qys2KsgsAKw9ZKupo76FCh
Base62	[Base62ieee]	F4bpVC8trxK13yapr3UFrY
Base62	[Base62sort]	5uRfL2yjhnArtoQfhtK5hO
Base64	[RFC4648], Section 4	wjKrAJQUEeyzyJ9r3s7YRg
Base64	[RFC4648], Section 5	wjKrAJQUEeyzyJ9r3s7YRg
Base64	[Base64sort]	kY9f-8FJ3Tmnm8xfrgvNGV
Base85	[Z85]	.znd(LOs.@V=F+O?P<+y

Table 19: UUID Test Vectors for RFC9562, Section A.1: UUIDv1

Encoding	Output
UUID-NCName-32	byizkwaeucqpmhse7nppm5wcgl
UUID-NCName-58	B6S7oX73gv2Y1iTENdXX8hL
UUID-NCName-64	BwjKrAJQUHsPIn2vezthGL
[Base62id]	LUZjlZguf8iMDOiFU7pUve

Table 20: UUID-NCName, Base62id Test Vectors for RFC9562, Section A.1: UUIDv1

C.5. UUIDv2

Encoding	Variant	DCE Security UUID
Base16	[RFC9562], Section 4	000003e8-cbb9-21ea-b201-00045a86c8a1
Base16	[RFC4648], Section 8	000003e8cbb921eab20100045a86c8a1
Base32	[RFC4648], Section 6	AAAAH2GLXEQ6VMQBAACFVBWIUE
Base32	[RFC4648], Section 7	00007Q6BN4GULCG10025L1M8K4
Base32	[Base32human]	00007T6BQ4GYNCG10025N1P8M4
Base36	---	5XJERAFS5KNNNG5WAM10H
Base52	---	KNcIMemTgumAjqCSfqp
Base58	[Base58btc]	2SMnXSegyRgxWPnMoHS
Base62	[Base62ieee]	azPmOJmh9xNaNgM2sh
Base62	[Base62sort]	azPmOJmh9xNaNgM2sh
Base64	[RFC4648], Section 4	AAAD6Mu5IeqyAQAEWobIoQ
Base64	[RFC4648], Section 5	AAAD6Mu5IeqyAQAEWobIoQ
Base64	[Base64sort]	---2uBit7Tem-F-3LcQ7cF
Base85	[Z85]	000b++EF!YVh<}dt87a(

Table 21: UUID Test Vectors for DCE Security: UUIDv2

Encoding	Output
UUID-NCName-32	caaaah2glxepkeaiaarninsfbl
UUID-NCName-58	C11KtP6Y9P3rRkvh2N1e__L
UUID-NCName-64	CAAAD6Mu5HqIBAARahsihL
[Base62id]	Fa84rLxnBVA2JNUzzkiHwn

Table 22: UUID-NCName, Base62id Test Vectors for DCE Security: UUIDv2

C.6. UUIDv3

Encoding	Variant	RFC9562, Section A.2
Base16	[RFC9562], Section 4	5df41881-3aed-3515-88a7-2f4a814cf09e
Base16	[RFC4648], Section 8	5df418813aed351588a72f4a814cf09e
Base32	[RFC4648], Section 6	LX2BRAJ25U2RLCFHF5FICTHQTY
Base32	[RFC4648], Section 7	BNQ1H09QTKQHB2575T582J7GJO
Base32	[Base32human]	BQT1H09TXMTHB2575X582K7GKR
Base36	---	5K8PHACEYCDEGBDB1VWP0EAHQ
Base52	---	CKwZBXIbBzTOnGTcpOSLRtq
Base58	[Base58btc]	CbuPE286MB6RsDazcU7sUy
Base62	[Base62ieee]	C1Rz9EB7xwwtaBE3hssbl8
Base62	[Base62sort]	2rHpz41xnmmjQ14tXiiRby
Base64	[RFC4648], Section 4	XfQYgTrtNRWIpy9KgUzwng
Base64	[RFC4648],	XfQYgTrtNRWIpy9KgUzwng

	Section 5	
Base64	[Base64sort]	MUFNVIfhCGL7dmx9VJnkbV
Base85	[Z85]	?1\tb3ped>Lo2muJP>R)

Table 23: UUID Test Vectors for RFC9562, Section A.2: UUIDv3

Encoding	Output
UUID-NCName-32	dlx2braj25vivrjzpjkauz4e6i
UUID-NCName-58	D3dTNMAmevR4NFAakRDtLdI
UUID-NCName-64	DXfQYgTrtUVinL0qBTPCeI
[Base62id]	IRPuPak8l8KDjbMTJxDqqE

Table 24: UUID-NCName, Base62id Test Vectors for RFC9562, Section A.2: UUIDv3

C.7. UUIDv4

Encoding	Variant	RFC9562, Section A.3
Base16	[RFC9562], Section 4	919108f7-52d1-4320-9bac-f847db4148a8
Base16	[RFC4648], Section 8	919108f752d143209bacf847db4148a8
Base32	[RFC4648], Section 6	SGIQR52S2FBSBG5M7BD5WQKIVA
Base32	[RFC4648], Section 7	I68GHTQIQ51I16TCV13TMGA8L0
Base32	[Base32human]	J68GHXTJT51J16XCZ13XPGA8N0
Base36	---	8M8SCPFUGFIJ4QENJ0IG77QG8
Base52	---	DWDhSoKPZRoqkgBWaJcOckY
Base58	[Base58btc]	JyZVoFVQxQNmw2bsgr7D1R

Base62	[Base62ieee]	EaqKcHGnV4iQSYo57bXn6o
Base62	[Base62sort]	2rHpz41xnmmjQ14tXiiRby
Base64	[RFC4648], Section 4	kZEI91LRQyCbrPhH20FIqA
Base64	[RFC4648], Section 5	kZEI91LRQyCbrPhH20FIqA
Base64	[Base64sort]	ZO37xpAGFm1QfEW6qo47e-
Base85	[Z85]	K=Y}zqQE&OO2(xP*D!xe

Table 25: UUID Test Vectors for RFC9562, Section A.3: UUIDv4

Encoding	Output
UUID-NCName-32	esgiqr52s2ezaxlhyi7nucsfij
UUID-NCName-58	E55CtqYNqva1mcmaa877eoJ
UUID-NCName-64	EkZEI91LRMgus-EfbQUioJ
[Base62id]	K0oEsdooJG5kbywVjft3Au

Table 26: UUID-NCName, Base62id Test Vectors for RFC9562, Section A.3: UUIDv4

C.8. UUIDv5

Encoding	Variant	RFC9562, Section A.4
Base16	[RFC9562], Section 4	2ed6657d-e927-568b-95e1-2665a8aea6a2
Base16	[RFC4648], Section 8	2ed6657de927568b95e12665a8aea6a2
Base32	[RFC4648], Section 6	F3LGK7PJE5LIXFPBEZS2RLVGUI
Base32	[RFC4648],	5RB6AVF94TB8N5F14PIQHBL6K8

	Section 7	
Base32	[Base32human]	5VB6AZF94XB8Q5F14SJTHBN6M8
Base36	---	2RTO2O5WTBFMSYWZX7KHQ1Z8I
Base52	---	BFPTsrUJhwfXFHQeVflYshu
Base58	[Base58btc]	6nTLogGvw2vmQjtATLqvLq
Base62	[Base62ieee]	BaXm0PEMkU5w7EKQaysNQO
Base62	[Base62sort]	1QNcqF4CaKvmx4AGQoiDGE
Base64	[RFC4648], Section 4	LtZlfeknVouV4SZlqK6mog
Base64	[RFC4648], Section 5	LtZlfeknVouV4SZlqK6mog
Base64	[Base64sort]	AhO_UTZbKciKsHO_e9uacV
Base85	[Z85]	f4KPZ>{GI&MeK63Sii1(

Table 27: UUID Test Vectors for RFC9562, Section A.4: UUIDv5

Encoding	Output
UUID-NCName-32	ff3lgk7pje5ullyjgmwuk5jvcj
UUID-NCName-58	F2K15VFLUBD326h169SNPjJ
UUID-NCName-64	FLtZlfeknaLXhJmWorqaiJ
[Base62id]	H0VhGlmNXgTHGeRqD3DcUU

Table 28: UUID-NCName, Base62id Test Vectors for RFC9562, Section A.4: UUIDv5

C.9. UUIDv6

Encoding	Variant	RFC9562, Section A.5
Base16	[RFC9562], Section 4	1EC9414C-232A-6B00-B3C8-9F6BDECED846
Base16	[RFC4648], Section 8	1EC9414C232A6B00B3C89F6BDECED846
Base32	[RFC4648], Section 6	D3EUCTBDFJVQBM6IT5V55TWYIY
Base32	[RFC4648], Section 7	3R4K2J1359LG1CU8JTLTTJMO8O
Base32	[Base32human]	3V4M2K1359NG1CY8KXNXXKPR8R
Base36	---	1TM3WVVTP7XVNZXA2TV43IJWM
Base52	---	liRiaXmgJTfBQppyCovyGu
Base58	[Base58btc]	4oVbpzb8BpnTH1mg11dmWd
Base62	[Base62ieee]	6FuGgfRpa7N6YbrlxT6FQ
Base62	[Base62sort]	w5k6WVHfQxDwORhbnJw5G
Base64	[RFC4648], Section 4	HslBTCMqawCzyJ9r3s7YRg
Base64	[RFC4648], Section 5	HslBTCMqawCzyJ9r3s7YRg
Base64	[Base64sort]	6g_0I1BePk1nm8xfrgvNGV
Base85	[Z85]	9)3YwbpWEeV=F+O?P<+y

Table 29: UUID Test Vectors for RFC9562, Section A.5: UUIDv6

Encoding	Output
UUID-NCName-32	gd3euctbdfkyahse7nppm5wcgl
UUID-NCName-58	GrxRCnDiX4mxSq8bFQjT3_L
UUID-NCName-64	GHslBTCMqsAPIn2vezthGL
[Base62id]	GWDoX3DScmUiFyjHO1pLJW

Table 30: UUID-NCName, Base62id Test Vectors for RFC9562, Section A.5: UUIDv6

C.10. UUIDv7

Encoding	Variant	RFC9562, Section A.6
Base16	[RFC9562], Section 4	017F22E2-79B0-7CC3-98C4-DC0C0C07398F
Base16	[RFC4648], Section 8	017F22E279B07CC398C4DC0C0C07398F
Base32	[RFC4648], Section 6	AF7SFYTZWB6MHGGE3QGAYBZZR4
Base32	[RFC4648], Section 7	05VI5OJPM1UC7664RG60O1PPHS
Base32	[Base32human]	05ZJ5RKSP1YC7664VG60R1SSHW
Base36	---	36TWI214QWJ7MGSVQ83NM8WF
Base52	---	BrKaFlCsyjaiCYuzuWvtun
Base58	[Base58btc]	BihbxwwQ4NZZpKRH9JDCz
Base62	[Base62ieee]	CzFyajyRd5A9oiF8QCBUD
Base62	[Base62sort]	2p5oQZoHTv0zeY5yG21K3
Base64	[RFC4648], Section 4	AX8i4nmwfMOYxNwMDAc5jw
Base64	[RFC4648],	AX8i4nmwfMOYxNwMDAc5jw

	Section 5	
Base64	[Base64sort]	-MwXsbakUBDNlCkB2-RtYk
Base85	[Z85]	0E(rMD9zXlN8E+*3<O!N

Table 31: UUID Test Vectors for RFC9562, Section A.6: UUIDv7

Encoding	Output
UUID-NCName-32	haf7sfytzwdgdrrg4bqgaoompj
UUID-NCName-58	H3RrXaX7uTM6qdwrXwpC6_J
UUID-NCName-64	HAX8i4nmwzDjE3AwMBzmPJ
[Base62id]	FcxAExHzEpSVJEpfkUXQYJ

Table 32: UUID-NCName, Base62id Test Vectors for RFC9562, Section A.6: UUIDv7

C.11. UUIDv8

Encoding	Variant	RFC9562, Section B.1
Base16	[RFC9562], Section 4	2489E9AD-2EE2-8E00-8EC9-32D5F69181C0
Base16	[RFC4648], Section 8	2489E9AD2EE28E008EC932D5F69181C0
Base32	[RFC4648], Section 6	ESE6TLJO4KHABDWJGLK7NEMBYA
Base32	[RFC4648], Section 7	4I4UJB9ESA7013M96BAVD4C1O0
Base32	[Base32human]	4J4YKB9EWA7013P96BAZD4C1R0
Base36	---	25VHD0YEN79F79OO0TV0BAE9S
Base52	---	skNtHBdXlnQCRPhYqjxSkQ
Base58	[Base58btc]	5WhDz2zW6g9mHu7EP9hoVq

Base62	[Base62ieee]	BG6ufXDWs4d4Dd5uoJIAi0
Base62	[Base62sort]	16wkVN3MiuTu3Tvke980Yq
Base64	[RFC4648], Section 4	JInprS7ijgCOyTLV9pGBwA
Base64	[RFC4648], Section 5	JInprS7ijgCOyTLV9pGBwA
Base64	[Base64sort]	87bdfHvXYV1DmIAKxd50k-
Base85	[Z85]	b-g=/f5?(>J(:6b{k%{w

Table 33: UUID Test Vectors for RFC9562, Section B.1: UUIDv8

Encoding	Output
UUID-NCName-32	iese6tljo4lqa5sjs2x3jdaoai
UUID-NCName-58	I22HpMAy5M181AjPFG7eLXI
UUID-NCName-64	IJInprS7i4A7JMtX2kYHAI
[Base62id]	Gh4ovtlXgG1ON4DKQNdPn6

Table 34: UUID-NCName, Base62id Test Vectors for RFC9562, Section B.1: UUIDv8

Encoding	Variant	RFC9562, Section B.2
Base16	[RFC9562], Section 4	5c146b14-3c52-8afd-938a-375d0df1fbf6
Base16	[RFC4648], Section 8	5c146b143c528afd938a375d0df1fbf6
Base32	[RFC4648], Section 6	LQKGWFB4KKFP3E4KG5OQ34P36Y
Base32	[RFC4648], Section 7	BGA6M51SAA5FR4SA6TEGRSFRUO
Base32	[Base32human]	BGA6P51WAA5FV4WA6XEGVWFVYR
Base36	---	5G8XXKU1AQGT02ZJGR8PA3EUU
Base52	---	CIhPGFswaUyeIiUVKFQScIW
Base58	[Base58btc]	CNV2iY4mKiTS1uw8RxapEH
Base62	[Base62ieee]	CxumvfWqhqp4Kq4BkCGR1s
Base62	[Base62sort]	2nkclVMgXgfuAgu1a26Hri
Base64	[RFC4648], Section 4	XBRrFDxSiv2TijddDfH79g
Base64	[RFC4648], Section 5	XBRrFDxSiv2TijddDfH79g
Base64	[Base64sort]	M0Gf42lHXjqIXYSS2U6vxV
Base85	[Z85]	tOH@/jw}7nLzRq84E%t?

Table 35: UUID Test Vectors for RFC9562, Section B.2: UUIDv8

Encoding	Output
UUID-NCName-32	ilqkgwfb4kkx5hcrxlug7d67wj
UUID-NCName-58	I3aR2J7aw1BJj4jJvfuWTXJ
UUID-NCName-64	IXBRrFDxSr9OKN10N8fv2J
[Base62id]	INshC24rV2DOUHBbMGbh5y

Table 36: UUID-NCName, Base62id Test Vectors for RFC9562, Section B.2: UUIDv8

C.12. Microsoft UUID

The following UUID 00000013-0000-0000-c000-000000000000 is for Microsoft.Azure.Portal and ensures this specification also includes non IETF/DCE variants of UUIDs.

Encoding	Variant	Microsoft.Azure.Portal
Base16	[RFC9562], Section 4	00000013-0000-0000-c000-000000000000
Base16	[RFC4648], Section 8	0000001300000000C000000000000000
Base32	[RFC4648], Section 6	AAAAAEYAAAAABQAAAAAAAAAAAA
Base32	[RFC4648], Section 7	000004O000001G000000000000
Base32	[Base32human]	000004R000001G000000000000
Base36	---	41Y09GGWTDKLN13WMO74
Base52	---	KGlWYrtEyPFBiZpPts
Base58	[Base58btc]	2anhPihXPV7NXZKLC7
Base62	[Base62ieee]	fjupi4ia0Gz3Pwu9y
Base62	[Base62sort]	VZkfYuYQq6ptFmkzo
Base64	[RFC4648], Section 4	AAAAEwAAAADAAAAAAAAAAA
Base64	[RFC4648], Section 5	AAAAEwAAAADAAAAAAAAAAA
Base64	[Base64sort]	----3k----2----------
Base85	[Z85]	0000j00000ZYjum00000

Table 37: UUID Test Vectors for Microsoft.Azure.Portal

Encoding	Output
UUID-NCName-32	aaaaaaeyaaaaaaaaaaaaaaaaam
UUID-NCName-58	A111Mo9hVUdmNWqcCExF__M
UUID-NCName-64	AAAAAEwAAAAAAAAAAAAAAM
[Base62id]	Fa84R2HvcuS2kQOPfUIAE4

Table 38: UUID-NCName, Base62id Test Vectors for Microsoft.Azure.Portal

Appendix D. Example UUID Base Encoding Tools, Libraries, and resources

The following list of libraries, tools, code, packages and other resources is for illustrative and research purposes.

Neither the authors or IETF endorse any of these libraries or guarantee their contents safe and harmless.

Install, download or use this software at your own risk.

D.1. Base32, Base

- https://github.com/chilts/sid
- https://www.jsdelivr.com/package/npm/uuid-encoder
- https://github.com/jetify-com/typeid
- https://docs.crunchybridge.com/api-concepts/eid
- https://hackage.haskell.org/package/ron-0.12/docs/RON-UUID.html
- https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/abencl_system_uuid.htm

D.2. Base32, Hex

- https://github.com/rs/xid

D.3. Base32, Crockford

- https://github.com/ulid/spec
- https://codeberg.org/prettyid/python
- https://ptrchm.com/posts/based-uuid/
- https://github.com/martinheidegger/uuid-b32
- https://docs.rs/fast32/latest/fast32/
- https://uuid.ramsey.dev/en/stable/rfc4122/version7.html
- https://crates.io/crates/crockford-uuid
- https://rymc.io/blog/2024/uuidv7-typeids-in-diesel/
- https://docs.rs/rusty_ulid/latest/rusty_ulid/

D.4. Base32, NCNAME

- https://www.rubydoc.info/gems/uuid-ncname

D.5. Base36

- https://github.com/paralleldrive/cuid2
- https://www.jsdelivr.com/package/npm/uuid-encoder
- https://duncan99.wordpress.com/2013/01/22/converting-uuid-to-base36/
- https://github.com/salieri/uuid-encoder
- https://stackoverflow.com/questions/62059588/shortening-a-guid
- https://gist.github.com/fabiolimace/508dd2dd9d32fd493b31a5f386d5d4bc
- https://classic.yarnpkg.com/en/package/base36-uuid

D.6. Base58

- https://github.com/cbschuld/uuid-base58
- https://www.linkedin.com/pulse/advantages-using-base58-unique-identifier-databases-lucian-ivanov
- https://github.com/AlexanderMatveev/go-uuid-base58
- https://packagist.org/packages/cbschuld/php-uuid-base58
- https://classic.yarnpkg.com/en/package/uuid58
- https://blog.schochastics.net/posts/2024-08-24_short-uuids/
- https://www.jsdelivr.com/package/npm/uuid-encoder

D.7. Base62

- https://github.com/boundary/flake
- https://github.com/segmentio/ksuid
- https://www.jsdelivr.com/package/npm/uuid-encoder
- https://grokipedia.com/page/Base62
- https://repost.aws/questions/QUPyiPyPsTTz6bnss1nQLoiQ/is-qldb-document-id-a-globally-unique-uuid
- https://hexdocs.pm/base62_uuid/readme.html
- https://github.com/lucasmichot/uuid62

D.8. Base64

- https://github.com/elastic/elasticsearch/blob/main/server/src/main/java/org/elasticsearch/common/UUIDs.java#L23
- https://github.com/chilts/sid
- https://github.com/twitter-archive/snowflake
- https://github.com/ppearcy/elasticflake/blob/master/src/main/java/org/limberware/elasticflake/Base64.java
- https://www.mongodb.com/docs/manual/reference/method/ObjectId.createFromBase64/
- https://firebase.blog/posts/2015/02/the-2120-ways-to-ensure-unique_68
- https://www.jsdelivr.com/package/npm/uuid-encoder
- https://base64-uuid.com/
- https://rcfed.com/Utilities/Base64GUID
- https://toolslick.com/conversion/data/guid
- https://guidgenerator.com/
- https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/abencl_system_uuid.htm

D.9. Base85

- https://stackoverflow.com/a/772984
- http://codehardblog.azurewebsites.net/encoding-uuid-based-keys-to-save-memory/
- https://gist.github.com/Higgs1/fee62d230bd87257e0b0
- https://www.npmjs.com/package/pure-uuid
- https://cjhaas.com/2013/11/12/php-base85-encode-128-bit-integer-guiduuid/
- https://webpowered.tools/textcodec/

Contributors

Brad G. Peabody
Email: brad@peabody.io

Author’s Address

Kyzer R. Davis
Cisco Systems
Email: kydavis@cisco.com, kyzer.davis@outlook.com