Stream: Internet Engineering Task Force (IETF)
RFC: 9562
Obsoletes: 4122
Category: Standards Track
Published: April 2024
ISSN: 2070-1721
Authors: K. Davis, *Cisco Systems*; B. Peabody, *Uncloud*; P. Leach, *University of Washington*

RFC 9562
Universally Unique IDentifiers (UUIDs)

Abstract

This specification defines UUIDs (Universally Unique IDentifiers) (also known as Globally Unique IDentifiers (GUIDs)) and a Uniform Resource Name namespace for UUIDs. A UUID is 128 bits long and is intended to guarantee uniqueness across space and time. UUIDs were originally used in the Apollo Network Computing System (NCS), later in the Open Software Foundation's (OSF's) Distributed Computing Environment (DCE), and then in Microsoft Windows platforms.

This specification is derived from the OSF DCE specification with the kind permission of the OSF (now known as "The Open Group"). Information from earlier versions of the OSF DCE specification have been incorporated into this document. This document obsoletes RFC 4122.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9562.

Copyright Notice

Copyright (c) 2024 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction

This specification defines a Uniform Resource Name namespace for Universally Unique IDentifiers (UUIDs) (also known as Globally Unique IDentifiers (GUIDs)). A UUID is 128 bits long and requires no central registration process.

The use of UUIDs is extremely pervasive in computing. They comprise the core identifier infrastructure for many operating systems such as Microsoft Windows and applications such as the Mozilla Web browser; in many cases, they can become exposed in many non-standard ways.

This specification attempts to standardize that practice as openly as possible and in a way that attempts to benefit the entire Internet. The information here is meant to be a concise guide for those wishing to implement services using UUIDs either in combination with URNs [RFC8141] or otherwise.

There is an ITU-T Recommendation and an ISO/IEC Standard [X667] that are derived from [RFC4122]. Both sets of specifications have been aligned and are fully technically compatible. Nothing in this document should be construed to override the DCE standards that defined UUIDs.

2. Motivation

One of the main reasons for using UUIDs is that no centralized authority is required to administer them (although two formats may leverage optional IEEE 802 Node IDs, others do not). As a result, generation on demand can be completely automated and used for a variety of purposes. The UUID generation algorithm described here supports very high allocation rates of 10 million per second per machine or more, if necessary, so that they could even be used as transaction IDs.

UUIDs are of a fixed size (128 bits), which is reasonably small compared to other alternatives. This lends itself well to sorting, ordering, and hashing of all sorts; storing in databases; simple allocation; and ease of programming in general.

Since UUIDs are unique and persistent, they make excellent URNs. The unique ability to generate a new UUID without a registration process allows for UUIDs to be one of the URNs with the lowest minting cost.

2.1. Update Motivation

Many things have changed in the time since UUIDs were originally created. Modern applications have a need to create and utilize UUIDs as the primary identifier for a variety of different items in complex computational systems, including but not limited to database keys, file names, machine or system names, and identifiers for event-driven transactions.

One area in which UUIDs have gained popularity is database keys. This stems from the increasingly distributed nature of modern applications. In such cases, "auto-increment" schemes that are often used by databases do not work well: the effort required to coordinate sequential

numeric identifiers across a network can easily become a burden. The fact that UUIDs can be used to create unique, reasonably short values in distributed systems without requiring coordination makes them a good alternative, but UUID versions 1-5, which were originally defined by [RFC4122], lack certain other desirable characteristics, such as:

1. UUID versions that are not time ordered, such as UUIDv4 (described in Section 5.4), have poor database-index locality. This means that new values created in succession are not close to each other in the index; thus, they require inserts to be performed at random locations. The resulting negative performance effects on the common structures used for this (B-tree and its variants) can be dramatic.
2. The 100-nanosecond Gregorian Epoch used in UUIDv1 timestamps (described in Section 5.1) is uncommon and difficult to represent accurately using a standard number format such as that described in [IEEE754].
3. Introspection/parsing is required to order by time sequence, as opposed to being able to perform a simple byte-by-byte comparison.
4. Privacy and network security issues arise from using a Media Access Control (MAC) address in the node field of UUIDv1. Exposed MAC addresses can be used as an attack surface to locate network interfaces and reveal various other information about such machines (minimally, the manufacturer and, potentially, other details). Additionally, with the advent of virtual machines and containers, uniqueness of the MAC address is no longer guaranteed.
5. Many of the implementation details specified in [RFC4122] involved trade-offs that are neither possible to specify for all applications nor necessary to produce interoperable implementations.
6. [RFC4122] did not distinguish between the requirements for generating a UUID and those for simply storing one, although they are often different.

Due to the aforementioned issues, many widely distributed database applications and large application vendors have sought to solve the problem of creating a better time-based, sortable unique identifier for use as a database key. This has led to numerous implementations over the past 10+ years solving the same problem in slightly different ways.

While preparing this specification, the following 16 different implementations were analyzed for trends in total ID length, bit layout, lexical formatting and encoding, timestamp type, timestamp format, timestamp accuracy, node format and components, collision handling, and multi-timestamp tick generation sequencing:

1. [ULID]
2. [LexicalUUID]
3. [Snowflake]
4. [Flake]
5. [ShardingID]
6. [KSUID]
7. [Elasticflake]
8. [FlakeID]

9. [Sonyflake]
10. [orderedUuid]
11. [COMBGUID]
12. [SID]
13. [pushID]
14. [XID]
15. [ObjectID]
16. [CUID]

An inspection of these implementations and the issues described above has led to this document, in which new UUIDs are adapted to address these issues.

Further, [RFC4122] itself was in need of an overhaul to address a number of topics such as, but not limited to, the following:

1. Implementation of miscellaneous errata reports. Mostly around bit-layout clarifications, which lead to inconsistent implementations [Err1957], [Err3546], [Err4976], etc.
2. Decoupling other UUID versions from the UUIDv1 bit layout so that fields like "time_hi_and_version" do not need to be referenced within a UUID that is not time based while also providing definition sections similar to that for UUIDv1 for UUIDv3, UUIDv4, and UUIDv5.
3. Providing implementation best practices around many real-world scenarios and corner cases observed by existing and prototype implementations.
4. Addressing security best practices and considerations for the modern age as it pertains to MAC addresses, hashing algorithms, secure randomness, and other topics.
5. Providing implementations a standard-based option for implementation-specific and/or experimental UUID designs.
6. Providing more test vectors that illustrate real UUIDs created as per the specification.

3. Terminology

3.1. Requirements Language

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.

3.2. Abbreviations

The following abbreviations are used in this document:

ABNF	Augmented Backus-Naur Form

CSPRNG	Cryptographically Secure Pseudorandom Number Generator
DBMS	Database Management System
IEEE	Institute of Electrical and Electronics Engineers
ITU	International Telecommunication Union
MAC	Media Access Control
MD5	Message Digest 5
MSB	Most Significant Bit
OID	Object Identifier
SHA	Secure Hash Algorithm
SHA-1	Secure Hash Algorithm 1 (with message digest of 160 bits)
SHA-3	Secure Hash Algorithm 3 (arbitrary size)
SHA-224	Secure Hash Algorithm 2 with message digest size of 224 bits
SHA-256	Secure Hash Algorithm 2 with message digest size of 256 bits
SHA-512	Secure Hash Algorithm 2 with message digest size of 512 bits
SHAKE	Secure Hash Algorithm 3 based on the KECCAK algorithm
URN	Uniform Resource Names
UTC	Coordinated Universal Time
UUID	Universally Unique Identifier
UUIDv1	Universally Unique Identifier version 1
UUIDv2	Universally Unique Identifier version 2
UUIDv3	Universally Unique Identifier version 3
UUIDv4	Universally Unique Identifier version 4
UUIDv5	Universally Unique Identifier version 5
UUIDv6	Universally Unique Identifier version 6
UUIDv7	Universally Unique Identifier version 7
UUIDv8	Universally Unique Identifier version 8

4. UUID Format

The UUID format is 16 octets (128 bits) in size; the variant bits in conjunction with the version bits described in the next sections determine finer structure. In terms of these UUID formats and layout, bit definitions start at 0 and end at 127, while octet definitions start at 0 and end at 15.

In the absence of explicit application or presentation protocol specification to the contrary, each field is encoded with the most significant byte first (known as "network byte order").

Saving UUIDs to binary format is done by sequencing all fields in big-endian format. However, there is a known caveat that Microsoft's Component Object Model (COM) GUIDs leverage little-endian when saving GUIDs. The discussion of this (see [MS_COM_GUID]) is outside the scope of this specification.

UUIDs **MAY** be represented as binary data or integers. When in use with URNs or as text in applications, any given UUID should be represented by the "hex-and-dash" string format consisting of multiple groups of uppercase or lowercase alphanumeric hexadecimal characters separated by single dashes/hyphens. When used with databases, please refer to Section 6.13.

The formal definition of the UUID string representation is provided by the following ABNF [RFC5234]:

```
UUID     = 4hexOctet "-"
           2hexOctet "-"
           2hexOctet "-"
           2hexOctet "-"
           6hexOctet
hexOctet = HEXDIG HEXDIG
DIGIT    = %x30-39
HEXDIG   = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
```

Note that the alphabetic characters may be all uppercase, all lowercase, or mixed case, as per Section 2.3 of [RFC5234]. An example UUID using this textual representation from the above ABNF is shown in Figure 1.

```
f81d4fae-7dec-11d0-a765-00a0c91e6bf6
```

Figure 1: Example String UUID Format

The same UUID from Figure 1 is represented in binary (Figure 2), as an unsigned integer (Figure 3), and as a URN (Figure 4) defined by [RFC8141].

```
11111000000111010100111110101110011111011110110000010001110100\
01010011101100101000000001010000011001001000111100110101111110110
```

Figure 2: Example Binary UUID

```
329800735698586629295641978511506172918
```

Figure 3: Example Unsigned Integer UUID (Shown as a Decimal Number)

```
urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6
```

Figure 4: Example URN Namespace for UUID

There are many other ways to define a UUID format; some examples are detailed below. Please note that this is not an exhaustive list and is only provided for informational purposes.

- Some UUID implementations, such as those found in [Python] and [Microsoft], will output UUID with the string format, including dashes, enclosed in curly braces.
- [X667] provides UUID format definitions for use of UUID with an OID.
- [IBM_NCS] is a legacy implementation that produces a unique UUID format compatible with Variant 0xx of Table 1.

4.1. Variant Field

The variant field determines the layout of the UUID. That is, the interpretation of all other bits in the UUID depends on the setting of the bits in the variant field. As such, it could more accurately be called a "type" field; we retain the original term for compatibility. The variant field consists of a variable number of the most significant bits of octet 8 of the UUID.

Table 1 lists the contents of the variant field, where the letter "x" indicates a "don't-care" value.

MSB0	MSB1	MSB2	MSB3	Variant	Description
0	x	x	x	1-7	Reserved. Network Computing System (NCS) backward compatibility, and includes Nil UUID as per Section 5.9.
1	0	x	x	8-9,A-B	The variant specified in this document.
1	1	0	x	C-D	Reserved. Microsoft Corporation backward compatibility.

MSB0	MSB1	MSB2	MSB3	Variant	Description
1	1	1	x	E-F	Reserved for future definition and includes Max UUID as per Section 5.10.

Table 1: UUID Variants

Interoperability, in any form, with variants other than the one defined here is not guaranteed but is not likely to be an issue in practice.

Specifically for UUIDs in this document, bits 64 and 65 of the UUID (bits 0 and 1 of octet 8) **MUST** be set to 1 and 0 as specified in row 2 of Table 1. Accordingly, all bit and field layouts avoid the use of these bits.

4.2. Version Field

The version number is in the most significant 4 bits of octet 6 (bits 48 through 51 of the UUID).

Table 2 lists all of the versions for this UUID variant 10xx specified in this document.

MSB0	MSB1	MSB2	MSB3	Version	Description
0	0	0	0	0	Unused.
0	0	0	1	1	The Gregorian time-based UUID specified in this document.
0	0	1	0	2	Reserved for DCE Security version, with embedded POSIX UUIDs.
0	0	1	1	3	The name-based version specified in this document that uses MD5 hashing.
0	1	0	0	4	The randomly or pseudorandomly generated version specified in this document.
0	1	0	1	5	The name-based version specified in this document that uses SHA-1 hashing.
0	1	1	0	6	Reordered Gregorian time-based UUID specified in this document.
0	1	1	1	7	Unix Epoch time-based UUID specified in this document.
1	0	0	0	8	Reserved for custom UUID formats specified in this document.
1	0	0	1	9	Reserved for future definition.

MSB0	MSB1	MSB2	MSB3	Version	Description
1	0	1	0	10	Reserved for future definition.
1	0	1	1	11	Reserved for future definition.
1	1	0	0	12	Reserved for future definition.
1	1	0	1	13	Reserved for future definition.
1	1	1	0	14	Reserved for future definition.
1	1	1	1	15	Reserved for future definition.

Table 2: UUID Variant 10xx Versions Defined by This Specification

An example version/variant layout for UUIDv4 follows the table where "M" represents the version placement for the hexadecimal representation of 0x4 (0b0100) and the "N" represents the variant placement for one of the four possible hexadecimal representation of variant 10xx: 0x8 (0b1000), 0x9 (0b1001), 0xA (0b1010), 0xB (0b1011).

```
00000000-0000-4000-8000-000000000000
00000000-0000-4000-9000-000000000000
00000000-0000-4000-A000-000000000000
00000000-0000-4000-B000-000000000000
xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx
```

Figure 5: UUIDv4 Variant Examples

It should be noted that the other remaining UUID variants found in Table 1 leverage different sub-typing or versioning mechanisms. The recording and definition of the remaining UUID variant and sub-typing combinations are outside of the scope of this document.

5. UUID Layouts

To minimize confusion about bit assignments within octets and among differing versions, the UUID record definition is provided as a grouping of fields within a bit layout consisting of four octets per row. The fields are presented with the most significant one first.

5.1. UUID Version 1

UUIDv1 is a time-based UUID featuring a 60-bit timestamp represented by Coordinated Universal Time (UTC) as a count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582 (the date of Gregorian reform to the Christian calendar).

UUIDv1 also features a clock sequence field that is used to help avoid duplicates that could arise when the clock is set backwards in time or if the Node ID changes.

The node field consists of an IEEE 802 MAC address, usually the host address or a randomly derived value per Sections 6.9 and 6.10.

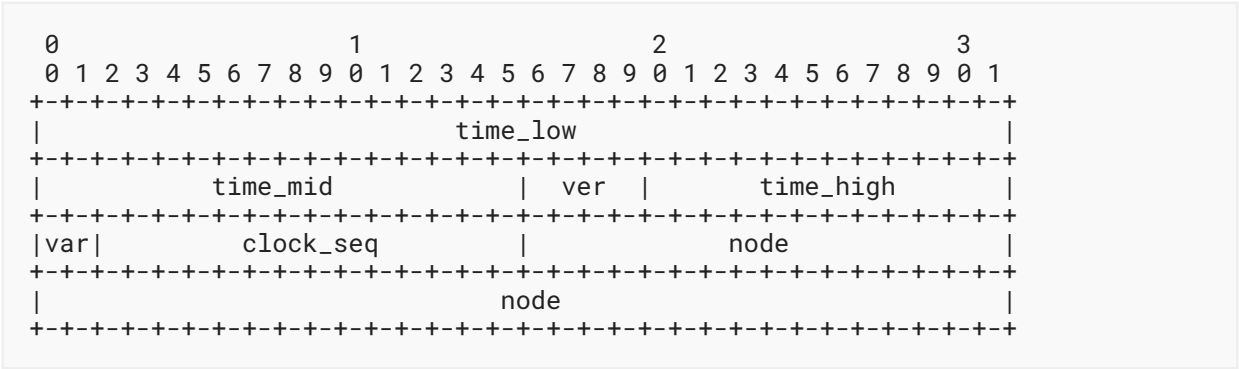

Figure 6: UUIDv1 Field and Bit Layout

time_low:
: The least significant 32 bits of the 60-bit starting timestamp. Occupies bits 0 through 31 (octets 0-3).

time_mid:
: The middle 16 bits of the 60-bit starting timestamp. Occupies bits 32 through 47 (octets 4-5).

ver:
: The 4-bit version field as defined by Section 4.2, set to 0b0001 (1). Occupies bits 48 through 51 of octet 6.

time_high:
: The least significant 12 bits from the 60-bit starting timestamp. Occupies bits 52 through 63 (octets 6-7).

var:
: The 2-bit variant field as defined by Section 4.1, set to 0b10. Occupies bits 64 and 65 of octet 8.

clock_seq:
: The 14 bits containing the clock sequence. Occupies bits 66 through 79 (octets 8-9).

node:
: 48-bit spatially unique identifier. Occupies bits 80 through 127 (octets 10-15).

For systems that do not have UTC available but do have the local time, they may use that instead of UTC as long as they do so consistently throughout the system. However, this is not recommended since generating the UTC from local time only needs a time-zone offset.

If the clock is set backwards, or if it might have been set backwards (e.g., while the system was powered off), and the UUID generator cannot be sure that no UUIDs were generated with timestamps larger than the value to which the clock was set, then the clock sequence **MUST** be changed. If the previous value of the clock sequence is known, it **MAY** be incremented; otherwise it **SHOULD** be set to a random or high-quality pseudorandom value.

Similarly, if the Node ID changes (e.g., because a network card has been moved between machines), setting the clock sequence to a random number minimizes the probability of a duplicate due to slight differences in the clock settings of the machines. If the value of the clock sequence associated with the changed Node ID were known, then the clock sequence **MAY** be incremented, but that is unlikely.

The clock sequence **MUST** be originally (i.e., once in the lifetime of a system) initialized to a random number to minimize the correlation across systems. This provides maximum protection against Node IDs that may move or switch from system to system rapidly. The initial value **MUST NOT** be correlated to the Node ID.

Notes about nodes derived from IEEE 802:

- On systems with multiple IEEE 802 addresses, any available one **MAY** be used.
- On systems with no IEEE address, a randomly or pseudorandomly generated value **MUST** be used; see Sections 6.9 and 6.10.
- On systems utilizing a 64-bit MAC address, the least significant, rightmost 48 bits **MAY** be used.
- Systems utilizing an IEEE 802.15.4 16-bit address **SHOULD** instead utilize their 64-bit MAC address where the least significant, rightmost 48 bits **MAY** be used. An alternative is to generate 32 bits of random data and postfix at the end of the 16-bit MAC address to create a 48-bit value.

5.2. UUID Version 2

UUIDv2 is for DCE Security UUIDs (see [C309] and [C311]). As such, the definition of these UUIDs is outside the scope of this specification.

5.3. UUID Version 3

UUIDv3 is meant for generating UUIDs from names that are drawn from, and unique within, some namespace as per Section 6.5.

UUIDv3 values are created by computing an MD5 hash [RFC1321] over a given Namespace ID value (Section 6.6) concatenated with the desired name value after both have been converted to a canonical sequence of octets, as defined by the standards or conventions of its namespace, in network byte order. This MD5 value is then used to populate all 128 bits of the UUID layout. The UUID version and variant then replace the respective bits as defined by Sections 4.2 and 4.1. An example of this bit substitution can be found in Appendix A.2.

Information around selecting a desired name's canonical format within a given namespace can be found in Section 6.5 under the heading "A note on names".

Where possible, UUIDv5 **SHOULD** be used in lieu of UUIDv3. For more information on MD5 security considerations, see [RFC6151].

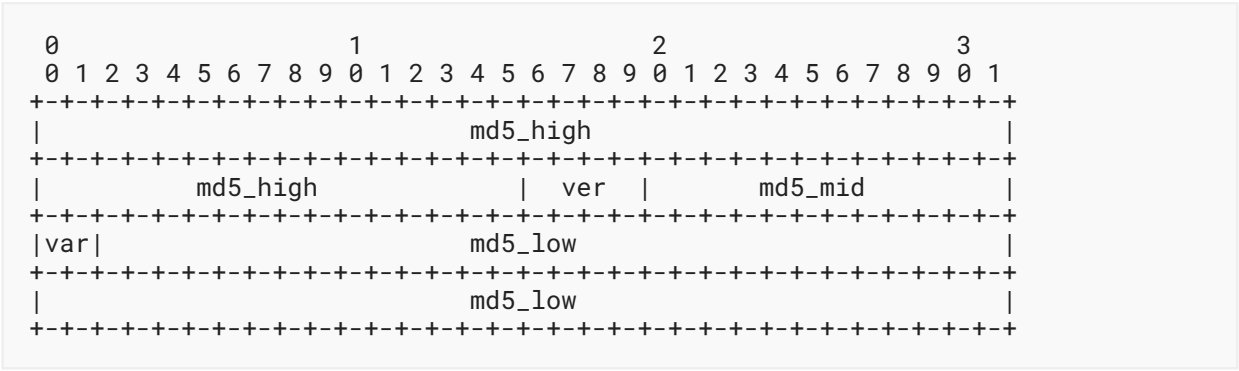

Figure 7: UUIDv3 Field and Bit Layout

md5_high:
: The first 48 bits of the layout are filled with the most significant, leftmost 48 bits from the computed MD5 value. Occupies bits 0 through 47 (octets 0-5).

ver:
: The 4-bit version field as defined by Section 4.2, set to 0b0011 (3). Occupies bits 48 through 51 of octet 6.

md5_mid:
: 12 more bits of the layout consisting of the least significant, rightmost 12 bits of 16 bits immediately following md5_high from the computed MD5 value. Occupies bits 52 through 63 (octets 6-7).

var:
: The 2-bit variant field as defined by Section 4.1, set to 0b10. Occupies bits 64 and 65 of octet 8.

md5_low:
: The final 62 bits of the layout immediately following the var field to be filled with the least significant, rightmost bits of the final 64 bits from the computed MD5 value. Occupies bits 66 through 127 (octets 8-15)

5.4. UUID Version 4

UUIDv4 is meant for generating UUIDs from truly random or pseudorandom numbers.

An implementation may generate 128 bits of random data that is used to fill out the UUID fields in Figure 8. The UUID version and variant then replace the respective bits as defined by Sections 4.1 and 4.2.

Alternatively, an implementation **MAY** choose to randomly generate the exact required number of bits for random_a, random_b, and random_c (122 bits total) and then concatenate the version and variant in the required position.

For guidelines on random data generation, see Section 6.9.

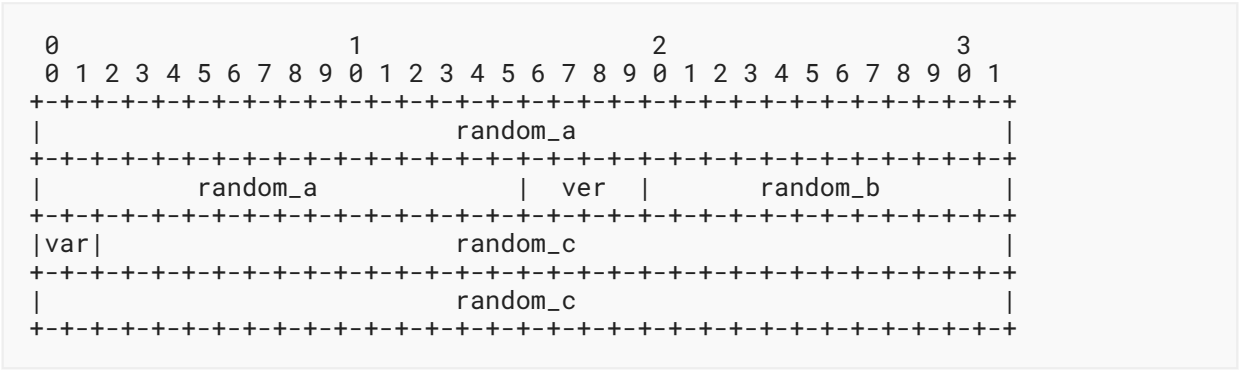

Figure 8: UUIDv4 Field and Bit Layout

random_a:
: The first 48 bits of the layout that can be filled with random data as specified in Section 6.9. Occupies bits 0 through 47 (octets 0-5).

ver:
: The 4-bit version field as defined by Section 4.2, set to 0b0100 (4). Occupies bits 48 through 51 of octet 6.

random_b:
: 12 more bits of the layout that can be filled random data as per Section 6.9. Occupies bits 52 through 63 (octets 6-7).

var:
: The 2-bit variant field as defined by Section 4.1, set to 0b10. Occupies bits 64 and 65 of octet 8.

random_c:
: The final 62 bits of the layout immediately following the var field to be filled with random data as per Section 6.9. Occupies bits 66 through 127 (octets 8-15).

5.5. UUID Version 5

UUIDv5 is meant for generating UUIDs from "names" that are drawn from, and unique within, some "namespace" as per Section 6.5.

UUIDv5 values are created by computing an SHA-1 hash [FIPS180-4] over a given Namespace ID value (Section 6.6) concatenated with the desired name value after both have been converted to a canonical sequence of octets, as defined by the standards or conventions of its namespace, in network byte order. The most significant, leftmost 128 bits of the SHA-1 value are then used to

populate all 128 bits of the UUID layout, and the remaining 32 least significant, rightmost bits of SHA-1 output are discarded. The UUID version and variant then replace the respective bits as defined by Sections 4.2 and 4.1. An example of this bit substitution and discarding excess bits can be found in Appendix A.4.

Information around selecting a desired name's canonical format within a given namespace can be found in Section 6.5 under the heading "A note on names".

There may be scenarios, usually depending on organizational security policies, where SHA-1 libraries may not be available or may be deemed unsafe for use. As such, it may be desirable to generate name-based UUIDs derived from SHA-256 or newer SHA methods. These name-based UUIDs **MUST NOT** utilize UUIDv5 and **MUST** be within the UUIDv8 space defined by Section 5.8. An illustrative example of UUIDv8 for SHA-256 name-based UUIDs is provided in Appendix B.2.

For more information on SHA-1 security considerations, see [RFC6194].

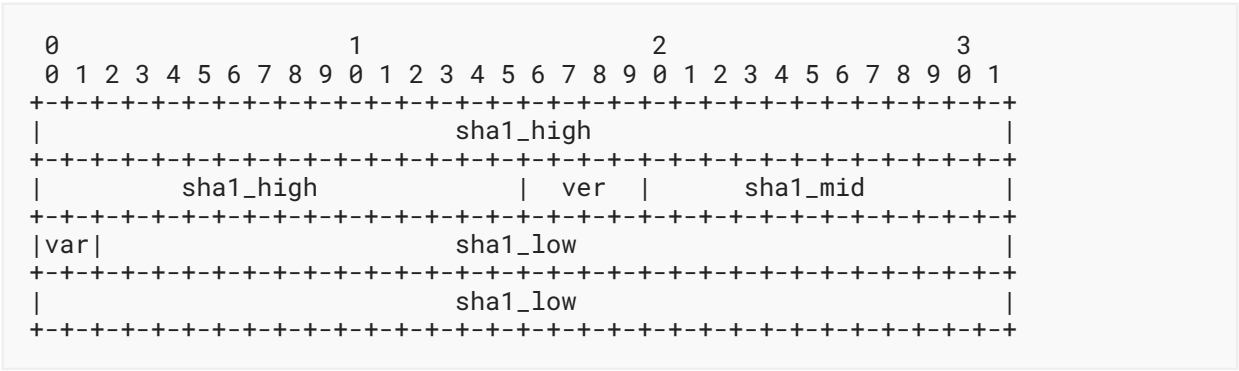

Figure 9: UUIDv5 Field and Bit Layout

sha1_high:
: The first 48 bits of the layout are filled with the most significant, leftmost 48 bits from the computed SHA-1 value. Occupies bits 0 through 47 (octets 0-5).

ver:
: The 4-bit version field as defined by Section 4.2, set to 0b0101 (5). Occupies bits 48 through 51 of octet 6.

sha1_mid:
: 12 more bits of the layout consisting of the least significant, rightmost 12 bits of 16 bits immediately following sha1_high from the computed SHA-1 value. Occupies bits 52 through 63 (octets 6-7).

var:
: The 2-bit variant field as defined by Section 4.1, set to 0b10. Occupies bits 64 and 65 of octet 8.

sha1_low:
: The final 62 bits of the layout immediately following the var field to be filled by skipping the two most significant, leftmost bits of the remaining SHA-1 hash and then using the next 62 most significant, leftmost bits. Any leftover SHA-1 bits are discarded and unused. Occupies bits 66 through 127 (octets 8-15).

5.6. UUID Version 6

UUIDv6 is a field-compatible version of UUIDv1 (Section 5.1), reordered for improved DB locality. It is expected that UUIDv6 will primarily be implemented in contexts where UUIDv1 is used. Systems that do not involve legacy UUIDv1 **SHOULD** use UUIDv7 (Section 5.7) instead.

Instead of splitting the timestamp into the low, mid, and high sections from UUIDv1, UUIDv6 changes this sequence so timestamp bytes are stored from most to least significant. That is, given a 60-bit timestamp value as specified for UUIDv1 in Section 5.1, for UUIDv6 the first 48 most significant bits are stored first, followed by the 4-bit version (same position), followed by the remaining 12 bits of the original 60-bit timestamp.

The clock sequence and node bits remain unchanged from their position in Section 5.1.

The clock sequence and node bits **SHOULD** be reset to a pseudorandom value for each new UUIDv6 generated; however, implementations **MAY** choose to retain the old clock sequence and MAC address behavior from Section 5.1. For more information on MAC address usage within UUIDs, see the Section 8.

The format for the 16-byte, 128-bit UUIDv6 is shown in Figure 10.

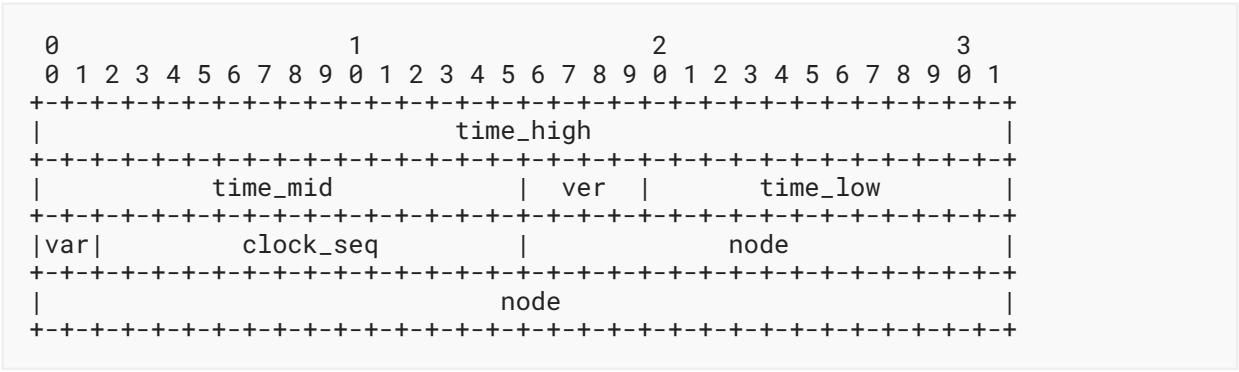

Figure 10: UUIDv6 Field and Bit Layout

time_high:
: The most significant 32 bits of the 60-bit starting timestamp. Occupies bits 0 through 31 (octets 0-3).

time_mid:
: The middle 16 bits of the 60-bit starting timestamp. Occupies bits 32 through 47 (octets 4-5).

ver:
: The 4-bit version field as defined by Section 4.2, set to 0b0110 (6). Occupies bits 48 through 51 of octet 6.

time_low:
: 12 bits that will contain the least significant 12 bits from the 60-bit starting timestamp. Occupies bits 52 through 63 (octets 6-7).

var:
: The 2-bit variant field as defined by Section 4.1, set to 0b10. Occupies bits 64 and 65 of octet 8.

clock_seq:
: The 14 bits containing the clock sequence. Occupies bits 66 through 79 (octets 8-9).

node:
: 48-bit spatially unique identifier. Occupies bits 80 through 127 (octets 10-15).

With UUIDv6, the steps for splitting the timestamp into time_high and time_mid are **OPTIONAL** since the 48 bits of time_high and time_mid will remain in the same order. An extra step of splitting the first 48 bits of the timestamp into the most significant 32 bits and least significant 16 bits proves useful when reusing an existing UUIDv1 implementation.

5.7. UUID Version 7

UUIDv7 features a time-ordered value field derived from the widely implemented and well-known Unix Epoch timestamp source, the number of milliseconds since midnight 1 Jan 1970 UTC, leap seconds excluded. Generally, UUIDv7 has improved entropy characteristics over UUIDv1 (Section 5.1) or UUIDv6 (Section 5.6).

UUIDv7 values are created by allocating a Unix timestamp in milliseconds in the most significant 48 bits and filling the remaining 74 bits, excluding the required version and variant bits, with random bits for each new UUIDv7 generated to provide uniqueness as per Section 6.9. Alternatively, implementations **MAY** fill the 74 bits, jointly, with a combination of the following subfields, in this order from the most significant bits to the least, to guarantee additional monotonicity within a millisecond:

1. An **OPTIONAL** sub-millisecond timestamp fraction (12 bits at maximum) as per Section 6.2 (Method 3).
2. An **OPTIONAL** carefully seeded counter as per Section 6.2 (Method 1 or 2).
3. Random data for each new UUIDv7 generated for any remaining space.

Implementations **SHOULD** utilize UUIDv7 instead of UUIDv1 and UUIDv6 if possible.

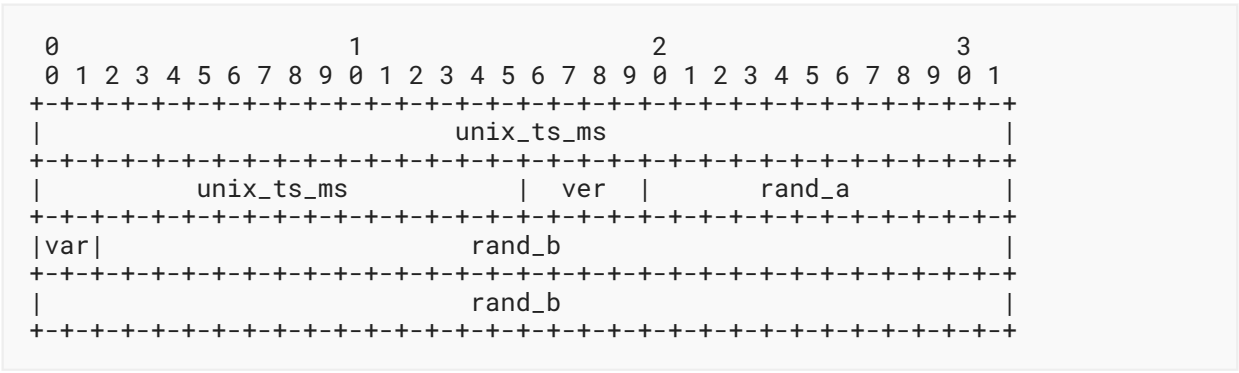

Figure 11: UUIDv7 Field and Bit Layout

unix_ts_ms:
: 48-bit big-endian unsigned number of the Unix Epoch timestamp in milliseconds as per Section 6.1. Occupies bits 0 through 47 (octets 0-5).

ver:
: The 4-bit version field as defined by Section 4.2, set to 0b0111 (7). Occupies bits 48 through 51 of octet 6.

rand_a:
: 12 bits of pseudorandom data to provide uniqueness as per Section 6.9 and/or optional constructs to guarantee additional monotonicity as per Section 6.2. Occupies bits 52 through 63 (octets 6-7).

var:
: The 2-bit variant field as defined by Section 4.1, set to 0b10. Occupies bits 64 and 65 of octet 8.

rand_b:
: The final 62 bits of pseudorandom data to provide uniqueness as per Section 6.9 and/or an optional counter to guarantee additional monotonicity as per Section 6.2. Occupies bits 66 through 127 (octets 8-15).

5.8. UUID Version 8

UUIDv8 provides a format for experimental or vendor-specific use cases. The only requirement is that the variant and version bits **MUST** be set as defined in Sections 4.1 and 4.2. UUIDv8's uniqueness will be implementation specific and **MUST NOT** be assumed.

The only explicitly defined bits are those of the version and variant fields, leaving 122 bits for implementation-specific UUIDs. To be clear, UUIDv8 is not a replacement for UUIDv4 (Section 5.4) where all 122 extra bits are filled with random data.

Some example situations in which UUIDv8 usage could occur:

- An implementation would like to embed extra information within the UUID other than what is defined in this document.
- An implementation has other application and/or language restrictions that inhibit the use of one of the current UUIDs.

Appendix B provides two illustrative examples of custom UUIDv8 algorithms to address two example scenarios.

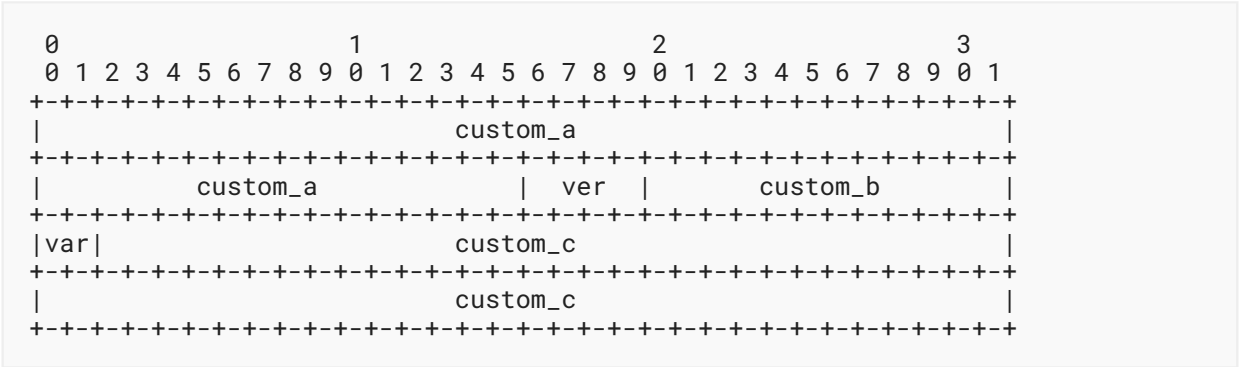

Figure 12: UUIDv8 Field and Bit Layout

custom_a:
: The first 48 bits of the layout that can be filled as an implementation sees fit. Occupies bits 0 through 47 (octets 0-5).

ver:
: The 4-bit version field as defined by Section 4.2, set to 0b1000 (8). Occupies bits 48 through 51 of octet 6.

custom_b:
: 12 more bits of the layout that can be filled as an implementation sees fit. Occupies bits 52 through 63 (octets 6-7).

var:
: The 2-bit variant field as defined by Section 4.1, set to 0b10. Occupies bits 64 and 65 of octet 8.

custom_c:
: The final 62 bits of the layout immediately following the var field to be filled as an implementation sees fit. Occupies bits 66 through 127 (octets 8-15).

5.9. Nil UUID

The Nil UUID is special form of UUID that is specified to have all 128 bits set to zero.

```
00000000-0000-0000-0000-000000000000
```

Figure 13: Nil UUID Format

A Nil UUID value can be useful to communicate the absence of any other UUID value in situations that otherwise require or use a 128-bit UUID. A Nil UUID can express the concept "no such value here". Thus, it is reserved for such use as needed for implementation-specific situations.

Note that the Nil UUID value falls within the range of the Apollo NCS variant as per the first row of Table 1 rather than the variant defined by this document.

5.10. Max UUID

The Max UUID is a special form of UUID that is specified to have all 128 bits set to 1. This UUID can be thought of as the inverse of the Nil UUID defined in Section 5.9.

```
FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF
```

Figure 14: Max UUID Format

A Max UUID value can be used as a sentinel value in situations where a 128-bit UUID is required, but a concept such as "end of UUID list" needs to be expressed and is reserved for such use as needed for implementation-specific situations.

Note that the Max UUID value falls within the range of the "yet-to-be defined" future UUID variant as per the last row of Table 1 rather than the variant defined by this document.

6. UUID Best Practices

The minimum requirements for generating UUIDs of each version are described in this document. Everything else is an implementation detail, and it is up to the implementer to decide what is appropriate for a given implementation. Various relevant factors are covered below to help guide an implementer through the different trade-offs among differing UUID implementations.

6.1. Timestamp Considerations

UUID timestamp source, precision, and length were topics of great debate while creating UUIDv7 for this specification. Choosing the right timestamp for your application is very important. This section will detail some of the most common points on this issue.

Reliability:
: Implementations acquire the current timestamp from a reliable source to provide values that are time ordered and continually increasing. Care must be taken to ensure that timestamp changes from the environment or operating system are handled in a way that is consistent with implementation requirements. For example, if it is possible for the system clock to move backward due to either manual adjustment or corrections from a time synchronization protocol, implementations need to determine how to handle such cases. (See "Altering, Fuzzing, or Smearing" below.)

Source:
: UUIDv1 and UUIDv6 both utilize a Gregorian Epoch timestamp, while UUIDv7 utilizes a Unix Epoch timestamp. If other timestamp sources or a custom timestamp Epoch are required, UUIDv8 **MUST** be used.

Sub-second Precision and Accuracy:
: Many levels of precision exist for timestamps: milliseconds, microseconds, nanoseconds, and beyond. Additionally, fractional representations of sub-second precision may be desired to mix various levels of precision in a time-ordered manner. Furthermore, system clocks themselves have an underlying granularity, which is frequently less than the precision offered by the operating system. With UUIDv1 and UUIDv6, 100 nanoseconds of precision are present, while UUIDv7 features a millisecond level of precision by default within the Unix Epoch that does not exceed the granularity capable in most modern systems. For other levels of precision, UUIDv8 is available. Similar to Section 6.2, with UUIDv1 or UUIDv6, a high-resolution timestamp can be simulated by keeping a count of the number of UUIDs that have been generated with the same value of the system time and using that count to construct the low order bits of the timestamp. The count of the high-resolution timestamp will range between zero and the number of 100-nanosecond intervals per system-time interval.

Length:
: The length of a given timestamp directly impacts how many timestamp ticks can be contained in a UUID before the maximum value for the timestamp field is reached. Take care to ensure that the proper length is selected for a given timestamp. UUIDv1 and UUIDv6 utilize a 60-bit timestamp valid until 5623 AD; UUIDv7 features a 48-bit timestamp valid until the year 10889 AD.

Altering, Fuzzing, or Smearing:
: Implementations **MAY** alter the actual timestamp. Some examples include security considerations around providing a real-clock value within a UUID to 1) correct inaccurate clocks, 2) handle leap seconds, or 3) obtain a millisecond value by dividing by 1024 (or some other value) for performance reasons (instead of dividing a number of microseconds by 1000). This specification makes no requirement or guarantee about how close the clock value needs to be to the actual time. If UUIDs do not need to be frequently generated, the UUIDv1 or UUIDv6 timestamp can simply be the system time multiplied by the number of 100-nanosecond intervals per system-time interval.

Padding:
: When timestamp padding is required, implementations **MUST** pad the most significant bits (leftmost) with data. An example for this padding data is to fill the most significant, leftmost bits of a Unix timestamp with zeroes to complete the 48-bit timestamp in UUIDv7. An alternative approach for padding data is to fill the most significant, leftmost bits with the number of 32-bit Unix timestamp rollovers after 2038-01-19.

Truncating:
: When timestamps need to be truncated, the lower, least significant bits **MUST** be used. An example would be truncating a 64-bit Unix timestamp to the least significant, rightmost 48 bits for UUIDv7.

Error Handling:
: If a system overruns the generator by requesting too many UUIDs within a single system-time interval, the UUID service can return an error or stall the UUID generator until the system clock catches up and **MUST NOT** knowingly return duplicate values due to a counter rollover. Note that if the processors overrun the UUID generation frequently, additional Node IDs can be allocated to the system, which will permit higher speed allocation by making multiple UUIDs potentially available for each timestamp value. Similar techniques are discussed in Section 6.4.

6.2. Monotonicity and Counters

Monotonicity (each subsequent value being greater than the last) is the backbone of time-based sortable UUIDs. Normally, time-based UUIDs from this document will be monotonic due to an embedded timestamp; however, implementations can guarantee additional monotonicity via the concepts covered in this section.

Take care to ensure UUIDs generated in batches are also monotonic. That is, if one thousand UUIDs are generated for the same timestamp, there should be sufficient logic for organizing the creation order of those one thousand UUIDs. Batch UUID creation implementations **MAY** utilize a monotonic counter that increments for each UUID created during a given timestamp.

For single-node UUID implementations that do not need to create batches of UUIDs, the embedded timestamp within UUIDv6 and UUIDv7 can provide sufficient monotonicity guarantees by simply ensuring that timestamp increments before creating a new UUID. Distributed nodes are discussed in Section 6.4.

Implementations **SHOULD** employ the following methods for single-node UUID implementations that require batch UUID creation or are otherwise concerned about monotonicity with high-frequency UUID generation.

Fixed Bit-Length Dedicated Counter (Method 1):
: Some implementations allocate a specific number of bits in the UUID layout to the sole purpose of tallying the total number of UUIDs created during a given UUID timestamp tick. If present, a fixed bit-length counter **MUST** be positioned immediately after the embedded timestamp. This promotes sortability and allows random data generation for each counter

increment. With this method, the rand_a section (or a subset of its leftmost bits) of UUIDv7 is used as a fixed bit-length dedicated counter that is incremented for every UUID generation. The trailing random bits generated for each new UUID in rand_b can help produce unguessable UUIDs. In the event that more counter bits are required, the most significant (leftmost) bits of rand_b **MAY** be used as additional counter bits.

Monotonic Random (Method 2):
: With this method, the random data is extended to also function as a counter. This monotonic value can be thought of as a "randomly seeded counter" that **MUST** be incremented in the least significant position for each UUID created on a given timestamp tick. UUIDv7's rand_b section **SHOULD** be utilized with this method to handle batch UUID generation during a single timestamp tick. The increment value for every UUID generation is a random integer of any desired length larger than zero. It ensures that the UUIDs retain the required level of unguessability provided by the underlying entropy. The increment value **MAY** be 1 when the number of UUIDs generated in a particular period of time is important and guessability is not an issue. However, incrementing the counter by 1 **SHOULD NOT** be used by implementations that favor unguessability, as the resulting values are easily guessable.

Replace Leftmost Random Bits with Increased Clock Precision (Method 3):
: For UUIDv7, which has millisecond timestamp precision, it is possible to use additional clock precision available on the system to substitute for up to 12 random bits immediately following the timestamp. This can provide values that are time ordered with sub-millisecond precision, using however many bits are appropriate in the implementation environment. With this method, the additional time precision bits **MUST** follow the timestamp as the next available bit in the rand_a field for UUIDv7.

  To calculate this value, start with the portion of the timestamp expressed as a fraction of the clock's tick value (fraction of a millisecond for UUIDv7). Compute the count of possible values that can be represented in the available bit space, 4096 for the UUIDv7 rand_a field. Using floating point or scaled integer arithmetic, multiply this fraction of a millisecond value by 4096 and round down (toward zero) to an integer result to arrive at a number between 0 and the maximum allowed for the indicated bits, which sorts monotonically based on time. Each increasing fractional value will result in an increasing bit field value to the precision available with these bits.

  For example, let's assume a system timestamp of 1 Jan 2023 12:34:56.1234567. Taking the precision greater than 1 ms gives us a value of 0.4567, as a fraction of a millisecond. If we wish to encode this as 12 bits, we can take the count of possible values that fit in those bits (4096 or 2^{12}), multiply it by our millisecond fraction value of 0.4567, and truncate the result to an integer, which gives an integer value of 1870. Expressed as hexadecimal, it is 0x74E or the binary bits 0b011101001110. One can then use those 12 bits as the most significant (leftmost) portion of the random section of the UUID (e.g., the rand_a field in UUIDv7). This works for any desired bit length that fits into a UUID, and applications can decide the appropriate length based on available clock precision; for UUIDv7, it is limited to 12 bits at maximum to reserve sufficient space for random bits.

The main benefit to encoding additional timestamp precision is that it utilizes additional time precision already available in the system clock to provide values that are more likely to be unique; thus, it may simplify certain implementations. This technique can also be used in conjunction with one of the other methods, where this additional time precision would immediately follow the timestamp. Then, if any bits are to be used as a clock sequence, they would follow next.

The following sub-topics cover issues related solely to creating reliable fixed bit-length dedicated counters:

Fixed Bit-Length Dedicated Counter Seeding:
: Implementations utilizing the fixed bit-length counter method randomly initialize the counter with each new timestamp tick. However, when the timestamp has not increased, the counter is instead incremented by the desired increment logic. When utilizing a randomly seeded counter alongside Method 1, the random value **MAY** be regenerated with each counter increment without impacting sortability. The downside is that Method 1 is prone to overflows if a counter of adequate length is not selected or the random data generated leaves little room for the required number of increments. Implementations utilizing fixed bit-length counter method **MAY** also choose to randomly initialize a portion of the counter rather than the entire counter. For example, a 24-bit counter could have the 23 bits in least significant, rightmost position randomly initialized. The remaining most significant, leftmost counter bit is initialized as zero for the sole purpose of guarding against counter rollovers.

Fixed Bit-Length Dedicated Counter Length:
: Select a counter bit-length that can properly handle the level of timestamp precision in use. For example, millisecond precision generally requires a larger counter than a timestamp with nanosecond precision. General guidance is that the counter **SHOULD** be at least 12 bits but no longer than 42 bits. Care must be taken to ensure that the counter length selected leaves room for sufficient entropy in the random portion of the UUID after the counter. This entropy helps improve the unguessability characteristics of UUIDs created within the batch.

The following sub-topics cover rollover handling with either type of counter method:

Counter Rollover Guards:
: The technique from "Fixed Bit-Length Dedicated Counter Seeding" above that describes allocating a segment of the fixed bit-length counter as a rollover guard is also helpful to mitigate counter rollover issues. This same technique can be used with monotonic random counter methods by ensuring that the total length of a possible increment in the least significant, rightmost position is less than the total length of the random value being incremented. As such, the most significant, leftmost bits can be incremented as rollover guarding.

Counter Rollover Handling:
: Counter rollovers **MUST** be handled by the application to avoid sorting issues. The general guidance is that applications that care about absolute monotonicity and sortability should freeze the counter and wait for the timestamp to advance, which ensures monotonicity is not broken. Alternatively, implementations **MAY** increment the timestamp ahead of the actual time and reinitialize the counter.

Implementations **MAY** use the following logic to ensure UUIDs featuring embedded counters are monotonic in nature:

1. Compare the current timestamp against the previously stored timestamp.
2. If the current timestamp is equal to the previous timestamp, increment the counter according to the desired method.
3. If the current timestamp is greater than the previous timestamp, re-initialize the desired counter method to the new timestamp and generate new random bytes (if the bytes were frozen or being used as the seed for a monotonic counter).

Monotonic Error Checking:
: Implementations **SHOULD** check if the currently generated UUID is greater than the previously generated UUID. If this is not the case, then any number of things could have occurred, such as clock rollbacks, leap second handling, and counter rollovers. Applications **SHOULD** embed sufficient logic to catch these scenarios and correct the problem to ensure that the next UUID generated is greater than the previous, or they should at least report an appropriate error. To handle this scenario, the general guidance is that the application **MAY** reuse the previous timestamp and increment the previous counter method.

6.3. UUID Generator States

The (optional) UUID generator state only needs to be read from stable storage once at boot time, if it is read into a system-wide shared volatile store (and updated whenever the stable store is updated).

This stable storage **MAY** be used to record various portions of the UUID generation, which prove useful for batch UUID generation purposes and monotonic error checking with UUIDv6 and UUIDv7. These stored values include but are not limited to last known timestamp, clock sequence, counters, and random data.

If an implementation does not have any stable store available, then it **MAY** proceed with UUID generation as if this were the first UUID created within a batch. This is the least desirable implementation because it will increase the frequency of creation of values such as clock sequence, counters, or random data, which increases the probability of duplicates. Further, frequent generation of random numbers also puts more stress on any entropy source and/or entropy pool being used as the basis for such random numbers.

An implementation **MAY** also return an application error in the event that collision resistance is of the utmost concern. The semantics of this error are up to the application and implementation. See Section 6.7 for more information on weighting collision tolerance in applications.

For UUIDv1 and UUIDv6, if the Node ID can never change (e.g., the network interface card from which the Node ID is derived is inseparable from the system), or if any change also re-initializes the clock sequence to a random value, then instead of keeping it in stable store, the current Node ID may be returned.

For UUIDv1 and UUIDv6, the state does not always need to be written to stable store every time a UUID is generated. The timestamp in the stable store can periodically be set to a value larger than any yet used in a UUID. As long as the generated UUIDs have timestamps less than that value, and the clock sequence and Node ID remain unchanged, only the shared volatile copy of the state needs to be updated. Furthermore, if the timestamp value in stable store is in the future by less than the typical time it takes the system to reboot, a crash will not cause a re-initialization of the clock sequence.

If it is too expensive to access shared state each time a UUID is generated, then the system-wide generator can be implemented to allocate a block of timestamps each time it is called; a per-process generator can allocate from that block until it is exhausted.

6.4. Distributed UUID Generation

Some implementations **MAY** desire the utilization of multi-node, clustered, applications that involve two or more nodes independently generating UUIDs that will be stored in a common location. While UUIDs already feature sufficient entropy to ensure that the chances of collision are low, as the total number of UUID generating nodes increases, so does the likelihood of a collision.

This section will detail the two additional collision resistance approaches that have been observed by multi-node UUID implementations in distributed environments.

It should be noted that, although this section details two methods for the sake of completeness, implementations should utilize the pseudorandom Node ID option if additional collision resistance for distributed UUID generation is a requirement. Likewise, utilization of either method is not required for implementing UUID generation in distributed environments.

Node IDs:

With this method, a pseudorandom Node ID value is placed within the UUID layout. This identifier helps ensure the bit space for a given node is unique, resulting in UUIDs that do not conflict with any other UUID created by another node with a different node id. Implementations that choose to leverage an embedded node id **SHOULD** utilize UUIDv8. The node id **SHOULD NOT** be an IEEE 802 MAC address per Section 8. The location and bit length are left to implementations and are outside the scope of this specification. Furthermore, the creation and negotiation of unique node ids among nodes is also out of scope for this specification.

Centralized Registry:
: With this method, all nodes tasked with creating UUIDs consult a central registry and confirm the generated value is unique. As applications scale, the communication with the central registry could become a bottleneck and impact UUID generation in a negative way. Shared knowledge schemes with central/global registries are outside the scope of this specification and are **NOT RECOMMENDED**.

Distributed applications generating UUIDs at a variety of hosts **MUST** be willing to rely on the random number source at all hosts.

6.5. Name-Based UUID Generation

Although some prefer to use the word "hash-based" to describe UUIDs featuring hashing algorithms (MD5 or SHA-1), this document retains the usage of the term "name-based" in order to maintain consistency with previously published documents and existing implementations.

The requirements for name-based UUIDs are as follows:

- UUIDs generated at different times from the same name (using the same canonical format) in the same namespace **MUST** be equal.
- UUIDs generated from two different names (same or differing canonical format) in the same namespace should be different (with very high probability).
- UUIDs generated from the same name (same or differing canonical format) in two different namespaces should be different (with very high probability).
- If two UUIDs that were generated from names (using the same canonical format) are equal, then they were generated from the same name in the same namespace (with very high probability).

A note on names:

> The concept of name (and namespace) should be broadly construed and not limited to textual names. A canonical sequence of octets is one that conforms to the specification for that name form's canonical representation. A name can have many usual forms, only one of which can be canonical. An implementer of new namespaces for UUIDs needs to reference the specification for the canonical form of names in that space or define such a canonical form for the namespace if it does not exist. For example, at the time of writing, Domain Name System (DNS) [RFC9499] has three conveyance formats: common (www.example.com), presentation (www.example.com.), and wire format (3www7example3com0). Looking at [X500] Distinguished Names (DNs), [RFC4122] allowed either text-based or binary DER-based names as inputs. For Uniform Resource Locators (URLs) [RFC1738], one could provide a Fully Qualified Domain Name (FQDN) with or without the protocol identifier www.example.com or https://www.example.com. When it comes to Object Identifiers (OIDs) [X660], one could choose dot notation without the leading dot (2.999), choose to include the leading dot (.2.999), or select one of the many formats from [X680] such as OID Internationalized Resource Identifier (OID-IRI) (/Joint-ISO-ITU-T/Example). While most users may default to the common format for DNS, FQDN format for a URL, text format for X.500, and dot notation without a leading dot for OID, name-based UUID implementations generally **SHOULD** allow arbitrary

input that will compute name-based UUIDs for any of the aforementioned example names and others not defined here. Each name format within a namespace will output different UUIDs. As such, the mechanisms or conventions used for allocating names and ensuring their uniqueness within their namespaces are beyond the scope of this specification.

6.6. Namespace ID Usage and Allocation

This section details the namespace IDs for some potentially interesting namespaces such as those for DNS [RFC9499], URLs [RFC1738], OIDs [X660], and DNs [X500].

Further, this section also details allocation, IANA registration, and other details pertinent to Namespace IDs.

Namespace	Namespace ID Value	Name Reference	Namespace ID Reference
DNS	6ba7b810-9dad-11d1-80b4-00c04fd430c8	[RFC9499]	[RFC4122], RFC 9562
URL	6ba7b811-9dad-11d1-80b4-00c04fd430c8	[RFC1738]	[RFC4122], RFC 9562
OID	6ba7b812-9dad-11d1-80b4-00c04fd430c8	[X660]	[RFC4122], RFC 9562
X500	6ba7b814-9dad-11d1-80b4-00c04fd430c8	[X500]	[RFC4122], RFC 9562

Table 3: Namespace IDs

Items may be added to this registry using the Specification Required policy as per [RFC8126].

For designated experts, generally speaking, Namespace IDs are allocated as follows:

- The first Namespace ID value, for DNS, was calculated from a time-based UUIDv1 and "6ba7b810-9dad-11d1-80b4-00c04fd430c8", used as a starting point.
- Subsequent Namespace ID values increment the least significant, rightmost bit of time_low "6ba7b810" while freezing the rest of the UUID to "9dad-11d1-80b4-00c04fd430c8".
- New Namespace ID values **MUST** use this same logic and **MUST NOT** use a previously used Namespace ID value.
- Thus, "6ba7b815" is the next available time_low for a new Namespace ID value with the full ID being "6ba7b815-9dad-11d1-80b4-00c04fd430c8".
- The upper bound for time_low in this special use, Namespace ID values, is "ffffffff" or "ffffffff-9dad-11d1-80b4-00c04fd430c8", which should be sufficient space for future Namespace ID values.

Note that the Namespace ID value "6ba7b813-9dad-11d1-80b4-00c04fd430c8" and its usage are not defined by this document or by [RFC4122]; thus, it **SHOULD NOT** be used as a Namespace ID value.

New Namespace ID values **MUST** be documented as per Section 7 if they are to be globally available and fully interoperable. Implementations **MAY** continue to use vendor-specific, application-specific, and deployment-specific Namespace ID values; but know that interoperability is not guaranteed. These custom Namespace ID values **MUST NOT** use the logic above; instead, generating a UUIDv4 or UUIDv7 Namespace ID value is **RECOMMENDED**. If collision probability (Section 6.7) and uniqueness (Section 6.8) of the final name-based UUID are not a problem, an implementation **MAY** also leverage UUIDv8 instead to create a custom, application-specific Namespace ID value.

Implementations **SHOULD** provide the ability to input a custom namespace to account for newly registered IANA Namespace ID values outside of those listed in this section or custom, application-specific Namespace ID values.

6.7. Collision Resistance

Implementations should weigh the consequences of UUID collisions within their application and when deciding between UUID versions that use entropy (randomness) versus the other components such as those in Sections 6.1 and 6.2. This is especially true for distributed node collision resistance as defined by Section 6.4.

There are two example scenarios below that help illustrate the varying seriousness of a collision within an application.

Low Impact:
: A UUID collision generated a duplicate log entry, which results in incorrect statistics derived from the data. Implementations that are not negatively affected by collisions may continue with the entropy and uniqueness provided by UUIDs defined in this document.

High Impact:
: A duplicate key causes an airplane to receive the wrong course, which puts people's lives at risk. In this scenario, there is no margin for error. Collisions must be avoided: failure is unacceptable. Applications dealing with this type of scenario must employ as much collision resistance as possible within the given application context.

6.8. Global and Local Uniqueness

UUIDs created by this specification **MAY** be used to provide local uniqueness guarantees. For example, ensuring UUIDs created within a local application context are unique within a database **MAY** be sufficient for some implementations where global uniqueness outside of the application context, in other applications, or around the world is not required.

Although true global uniqueness is impossible to guarantee without a shared knowledge scheme, a shared knowledge scheme is not required by a UUID to provide uniqueness for practical implementation purposes. Implementations **MAY** use a shared knowledge scheme, introduced in Section 6.4, as they see fit to extend the uniqueness guaranteed by this specification.

6.9. Unguessability

Implementations **SHOULD** utilize a cryptographically secure pseudorandom number generator (CSPRNG) to provide values that are both difficult to predict ("unguessable") and have a low likelihood of collision ("unique"). The exception is when a suitable CSPRNG is unavailable in the execution environment. Take care to ensure the CSPRNG state is properly reseeded upon state changes, such as process forks, to ensure proper CSPRNG operation. CSPRNG ensures the best of Sections 6.7 and 8 are present in modern UUIDs.

Further advice on generating cryptographic-quality random numbers can be found in [RFC4086], [RFC8937], and [RANDOM].

6.10. UUIDs That Do Not Identify the Host

This section describes how to generate a UUIDv1 or UUIDv6 value if an IEEE 802 address is not available or its use is not desired.

Implementations **MAY** leverage MAC address randomization techniques [IEEE802.11bh] as an alternative to the pseudorandom logic provided in this section.

Alternatively, implementations **MAY** elect to obtain a 48-bit cryptographic-quality random number as per Section 6.9 to use as the Node ID. After generating the 48-bit fully randomized node value, implementations **MUST** set the least significant bit of the first octet of the Node ID to 1. This bit is the unicast or multicast bit, which will never be set in IEEE 802 addresses obtained from network cards. Hence, there can never be a conflict between UUIDs generated by machines with and without network cards. An example of generating a randomized 48-bit node value and the subsequent bit modification is detailed in Appendix A. For more information about IEEE 802 address and the unicast or multicast or local/global bits, please review [RFC9542].

For compatibility with earlier specifications, note that this document uses the unicast or multicast bit instead of the arguably more correct local/global bit because MAC addresses with the local/global bit set or not set are both possible in a network. This is not the case with the unicast or multicast bit. One node cannot have a MAC address that multicasts to multiple nodes.

In addition, items such as the computer's name and the name of the operating system, while not strictly speaking random, will help differentiate the results from those obtained by other systems.

The exact algorithm to generate a Node ID using these data is system specific because both the data available and the functions to obtain them are often very system specific. However, a generic approach is to accumulate as many sources as possible into a buffer, use a message digest (such as SHA-256 or SHA-512 defined by [FIPS180-4]), take an arbitrary 6 bytes from the hash value, and set the multicast bit as described above.

6.11. Sorting

UUIDv6 and UUIDv7 are designed so that implementations that require sorting (e.g., database indexes) sort as opaque raw bytes without the need for parsing or introspection.

Time-ordered monotonic UUIDs benefit from greater database-index locality because the new values are near each other in the index. As a result, objects are more easily clustered together for better performance. The real-world differences in this approach of index locality versus random data inserts can be one order of magnitude or more.

UUID formats created by this specification are intended to be lexicographically sortable while in the textual representation.

UUIDs created by this specification are crafted with big-endian byte order (network byte order) in mind. If little-endian style is required, UUIDv8 is available for custom UUID formats.

6.12. Opacity

As general guidance, avoiding parsing UUID values unnecessarily is recommended; instead, treat UUIDs as opaquely as possible. Although application-specific concerns could, of course, require some degree of introspection (e.g., to examine Sections 4.1 or 4.2 or perhaps the timestamp of a UUID), the advice here is to avoid this or other parsing unless absolutely necessary. Applications typically tend to be simpler, be more interoperable, and perform better when this advice is followed.

6.13. DBMS and Database Considerations

For many applications, such as databases, storing UUIDs as text is unnecessarily verbose, requiring 288 bits to represent 128-bit UUID values. Thus, where feasible, UUIDs **SHOULD** be stored within database applications as the underlying 128-bit binary value.

For other systems, UUIDs **MAY** be stored in binary form or as text, as appropriate. The trade-offs to both approaches are as follows:

- Storing in binary form requires less space and may result in faster data access.
- Storing as text requires more space but may require less translation if the resulting text form is to be used after retrieval, which may make it simpler to implement.

DBMS vendors are encouraged to provide functionality to generate and store UUID formats defined by this specification for use as identifiers or left parts of identifiers such as, but not limited to, primary keys, surrogate keys for temporal databases, foreign keys included in polymorphic relationships, and keys for key-value pairs in JSON columns and key-value databases. Applications using a monolithic database may find using database-generated UUIDs (as opposed to client-generated UUIDs) provides the best UUID monotonicity. In addition to UUIDs, additional identifiers **MAY** be used to ensure integrity and feedback.

Designers of database schema are cautioned against using name-based UUIDs (see Sections 5.3 and 5.5) as primary keys in tables. A common issue observed in database schema design is the assumption that a particular value will never change, which later turns out to be an incorrect assumption. Postal codes, license or other identification numbers, and numerous other such identifiers seem unique and unchanging at a given point time -- only later to have edge cases where they need to change. The subsequent change of the identifier, used as a "name" input for name-based UUIDs, can invalidate a given database structure. In such scenarios, it is observed that using any non-name-based UUID version would have resulted in the field in question being placed somewhere that would have been easier to adapt to such changes (primary key excluded from this statement). The general advice is to avoid name-based UUID natural keys and, instead, to utilize time-based UUID surrogate keys based on the aforementioned problems detailed in this section.

7. IANA Considerations

All references to [RFC4122] in IANA registries (outside of those created by this document) have been replaced with references to this document, including the IANA URN namespace registration [URNNamespaces] for UUID. References to Section 4.1.2 of [RFC4122] have been updated to refer to Section 4 of this document.

Finally, IANA should track UUID Subtypes and Special Case "Namespace IDs Values" as specified in Sections 7.1 and 7.2 at the following location: <https://www.iana.org/assignments/uuid>.

When evaluating requests, the designated expert should consider community feedback, how well-defined the reference specification is, and this specification's requirements. Vendor-specific, application-specific, and deployment-specific values are unable to be registered. Specification documents should be published in a stable, freely available manner (ideally, located with a URL) but need not be standards. The designated expert will either approve or deny the registration request and communicate this decision to IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

7.1. IANA UUID Subtype Registry and Registration

This specification defines the "UUID Subtypes" registry for common widely used UUID standards.

Name	ID	Subtype	Variant	Reference
Gregorian Time-based	1	version	OSF DCE / IETF	[RFC4122], RFC 9562
DCE Security	2	version	OSF DCE / IETF	[C309], [C311]
MD5 Name-based	3	version	OSF DCE / IETF	[RFC4122], RFC 9562
Random	4	version	OSF DCE / IETF	[RFC4122], RFC 9562
SHA-1 Name-based	5	version	OSF DCE / IETF	[RFC4122], RFC 9562

Name	ID	Subtype	Variant	Reference
Reordered Gregorian Time-based	6	version	OSF DCE / IETF	RFC 9562
Unix Time-based	7	version	OSF DCE / IETF	RFC 9562
Custom	8	version	OSF DCE / IETF	RFC 9562

Table 4: IANA UUID Subtypes

This table may be extended by Standards Action as per [RFC8126].

For designated experts:

- The minimum and maximum "ID" value for the subtype "version" within the "OSF DCE / IETF" variant is 0 through 15. The versions within Table 1 described as "Reserved for future definition" or "unused" are omitted from this IANA registry until properly defined.
- The "Subtype" column is free-form text. However, at the time of publication, "version" and "family" are the only known UUID subtypes. The "family" subtype is part of the "Apollo NCS" variant space (both are outside the scope of this specification). The Microsoft variant may have subtyping mechanisms defined; however, they are unknown and outside of the scope of this specification. Similarly, the final "Reserved for future definition" variant may introduce new subtyping logic at a future date. Subtype IDs are permitted to overlap. That is, an ID of "1" may exist in multiple variant spaces.
- The "Variant" column is free-form text. However, it is likely that one of four values will be included: the first three are "OSF DCE / IETF", "Apollo NCS", and "Microsoft", and the final variant value belongs to the "Reserved for future definition" variant and may introduce a new name at a future date.

7.2. IANA UUID Namespace ID Registry and Registration

This specification defines the "UUID Namespace IDs" registry for common, widely used Namespace ID values.

The full details of this registration, including information for designated experts, can be found in Section 6.6.

8. Security Considerations

Implementations **SHOULD NOT** assume that UUIDs are hard to guess. For example, they **MUST NOT** be used as security capabilities (identifiers whose mere possession grants access). Discovery of predictability in a random number source will result in a vulnerability.

Implementations **MUST NOT** assume that it is easy to determine if a UUID has been slightly modified in order to redirect a reference to another object. Humans do not have the ability to easily check the integrity of a UUID by simply glancing at it.

MAC addresses pose inherent security risks around privacy and **SHOULD NOT** be used within a UUID. Instead CSPRNG data **SHOULD** be selected from a source with sufficient entropy to ensure guaranteed uniqueness among UUID generation. See Sections 6.9 and 6.10 for more information.

Timestamps embedded in the UUID do pose a very small attack surface. The timestamp in conjunction with an embedded counter does signal the order of creation for a given UUID and its corresponding data but does not define anything about the data itself or the application as a whole. If UUIDs are required for use with any security operation within an application context in any shape or form, then UUIDv4 (Section 5.4) **SHOULD** be utilized.

See [RFC6151] for MD5 security considerations and [RFC6194] for SHA-1 security considerations.

9. References

9.1. Normative References

[C309] X/Open Company Limited, "X/Open DCE: Remote Procedure Call", ISBN 1-85912-041-5, Open CAE Specification C309, August 1994, <https://pubs.opengroup.org/onlinepubs/9696999099/toc.pdf>.

[C311] The Open Group, "DCE 1.1: Authentication and Security Services", Open Group CAE Specification C311, August 1997, <https://pubs.opengroup.org/onlinepubs/9696989899/toc.pdf>.

[FIPS180-4] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", FIPS PUB 180-4, DOI 10.6028/NIST.FIPS.180-4, August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

[FIPS202] National Institute of Standards and Technology (NIST), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", FIPS PUB 202, DOI 10.6028/NIST.FIPS.202, August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>.

[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/info/rfc2119>.

[RFC8141] Saint-Andre, P. and J. Klensin, "Uniform Resource Names (URNs)", RFC 8141, DOI 10.17487/RFC8141, April 2017, <https://www.rfc-editor.org/info/rfc8141>.

[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/info/rfc8174>.

[X667] ITU-T, "Information technology - Open Systems Interconnection - Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 object identifier components", ISO/IEC 9834-8:2004, ITU-T Recommendation X.667, September 2004.

9.2. Informative References

[COMBGUID] "Creating sequential GUIDs in C# for MSSQL or PostgreSql", commit 2759820, December 2020, <https://github.com/richardtallent/RT.Comb>.

[CUID] "Collision-resistant ids optimized for horizontal scaling and performance.", commit 215b27b, October 2020, <https://github.com/ericelliott/cuid>.

[Elasticflake] Pearcy, P., "Sequential UUID / Flake ID generator pulled out of elasticsearch common", commit dd71c21, January 2015, <https://github.com/ppearcy/elasticflake>.

[Err1957] RFC Errata, Erratum ID 1957, RFC 4122, <https://www.rfc-editor.org/errata/eid1957>.

[Err3546] RFC Errata, Erratum ID 3546, RFC 4122, <https://www.rfc-editor.org/errata/eid3546>.

[Err4976] RFC Errata, Erratum ID 4976, RFC 4122, <https://www.rfc-editor.org/errata/eid4976>.

[Flake] Boundary, "Flake: A decentralized, k-ordered id generation service in Erlang", commit 15c933a, February 2017, <https://github.com/boundary/flake>.

[FlakeID] "Flake ID Generator", commit fcd6a2f, April 2020, <https://github.com/T-PWK/flake-idgen>.

[IBM_NCS] IBM, "uuid_gen Command (NCS)", March 2023, <https://www.ibm.com/docs/en/aix/7.1?topic=u-uuid-gen-command-ncs>.

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic.", IEEE Std 754-2019, DOI 10.1109/IEEESTD.2019.8766229, July 2019, <https://standards.ieee.org/ieee/754/6210/>.

[IEEE802.11bh] IEEE, "IEEE Draft Standard for Information technology--Telecommunications and information exchange between systems Local and metropolitan area networks--Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment: Enhancements for Extremely High Throughput (EHT)", Electronic ISBN 978-1-5044-9520-2, March 2023, <https://standards.ieee.org/ieee/802.11bh/10525/>.

[KSUID] Segment, "K-Sortable Globally Unique IDs", commit bf376a7, July 2020, <https://github.com/segmentio/ksuid>.

[LexicalUUID] Twitter, "Cassie", commit f6da4e0, November 2012, <https://github.com/twitter-archive/cassie>.

[Microsoft] Microsoft, "2.3.4.3 GUID - Curly Braced String Representation", April 2023, <https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-dtyp/222af2d3-5c00-4899-bc87-ed4c6515e80d>.

[MS_COM_GUID] Chen, R., "Why does COM express GUIDs in a mix of big-endian and little-endian? Why can't it just pick a side and stick with it?", September 2022, <https://devblogs.microsoft.com/oldnewthing/20220928-00/?p=107221>.

[ObjectID] MongoDB, "ObjectId", <https://docs.mongodb.com/manual/reference/method/ObjectId/>.

[orderedUuid] Cabrera, I. B., "Laravel: The mysterious "Ordered UUID"", January 2020, <https://itnext.io/laravel-the-mysterious-ordered-uuid-29e7500b4f8>.

[pushID] Lehenbauer, M., "The 2^120 Ways to Ensure Unique Identifiers", February 2015, <https://firebase.googleblog.com/2015/02/the-2120-ways-to-ensure-unique_68.html>.

[Python] Python, "uuid - UUID objects according to RFC 4122", <https://docs.python.org/3/library/uuid.html>.

[RANDOM] Occil, P., "Random Number Generator Recommendations for Applications", June 2023, <https://peteroupc.github.io/random.html>.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, DOI 10.17487/RFC1321, April 1992, <https://www.rfc-editor.org/info/rfc1321>.

[RFC1738] Berners-Lee, T., Masinter, L., and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, DOI 10.17487/RFC1738, December 1994, <https://www.rfc-editor.org/info/rfc1738>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <https://www.rfc-editor.org/info/rfc6151>.

[RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, DOI 10.17487/RFC6194, March 2011, <https://www.rfc-editor.org/info/rfc6194>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8937] Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N., and C. Wood, "Randomness Improvements for Security Protocols", RFC 8937, DOI 10.17487/RFC8937, October 2020, <https://www.rfc-editor.org/info/rfc8937>.

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

[RFC9542] Eastlake 3rd, D., Abley, J., and Y. Li, "IANA Considerations and IETF Protocol and Documentation Usage for IEEE 802 Parameters", BCP 141, RFC 9542, DOI 10.17487/RFC9542, April 2024, <https://www.rfc-editor.org/info/rfc9542>.

[ShardingID] Instagram Engineering, "Sharding & IDs at Instagram", December 2012, <https://instagram-engineering.com/sharding-ids-at-instagram-1cf5a71e5a5c>.

[SID] "sid : generate sortable identifiers", Commit 660e947, June 2019, <https://github.com/chilts/sid>.

[Snowflake] Twitter, "Snowflake is a network service for generating unique ID numbers at high scale with some simple guarantees.", commit ec40836, May 2014, <https://github.com/twitter-archive/snowflake>.

[Sonyflake] Sony, "A distributed unique ID generator inspired by Twitter's Snowflake", commit 848d664, August 2020, <https://github.com/sony/sonyflake>.

[ULID] "Universally Unique Lexicographically Sortable Identifier", Commit d0c7170, May 2019, <https://github.com/ulid/spec>.

[URNNamespaces] IANA, "Uniform Resource Names (URN) Namespaces", <https://www.iana.org/assignments/urn-namespaces/>.

[X500] ITU-T, "Information technology - Open Systems Interconnection - The Directory: Overview of concepts, models and services", ISO/IEC 9594-1, ITU-T Recommendation X.500, October 2019.

[X660] ITU-T, "Information technology - Procedures for the operation of object identifier registration authorities: General procedures and top arcs of the international object identifier tree", ISO/IEC 9834-1, ITU-T Recommendation X.660, July 2011.

[X680] ITU-T, "Information Technology - Abstract Syntax Notation One (ASN.1) & ASN.1 encoding rules", ISO/IEC 8824-1:2021, ITU-T Recommendation X.680, February 2021.

[XID] "Globally Unique ID Generator", commit efa678f, October 2020, <https://github.com/rs/xid>.

Appendix A. Test Vectors

Both UUIDv1 and UUIDv6 test vectors utilize the same 60-bit timestamp: 0x1EC9414C232AB00 (138648505420000000) Tuesday, February 22, 2022 2:22:22.000000 PM GMT-05:00.

Both UUIDv1 and UUIDv6 utilize the same values in clock_seq and node; all of which have been generated with random data. For the randomized node, the least significant bit of the first octet is set to a value of 1 as per Section 6.10. Thus, the starting value 0x9E6BDECED846 was changed to 0x9F6BDECED846.

The pseudocode used for converting from a 64-bit Unix timestamp to a 100 ns Gregorian timestamp value has been left in the document for reference purposes.

```
# Gregorian-to-Unix Offset:
# The number of 100 ns intervals between the
# UUID Epoch 1582-10-15 00:00:00
# and the Unix Epoch 1970-01-01 00:00:00
# Greg_Unix_offset = 0x01b21dd213814000 or 122192928000000000

# Unix 64-bit Nanosecond Timestamp:
# Unix NS: Tuesday, February 22, 2022 2:22:22 PM GMT-05:00
# Unix_64_bit_ns = 0x16D6320C3D4DCC00 or 1645557742000000000

# Unix Nanosecond precision to Gregorian 100-nanosecond intervals
# Greg_100_ns = (Unix_64_bit_ns/100)+Greg_Unix_offset

# Work:
# Greg_100_ns = (1645557742000000000/100)+122192928000000000
# Unix_64_bit_ns = (138648505420000000-122192928000000000)*100

# Final:
# Greg_100_ns = 0x1EC9414C232AB00 or 138648505420000000
```

Figure 15: Test Vector Timestamp Pseudocode

A.1. Example of a UUIDv1 Value

```
-----------------------------------------
field      bits value
-----------------------------------------
time_low   32   0xC232AB00
time_mid   16   0x9414
ver         4   0x1
time_high  12   0x1EC
var         2   0b10
clock_seq  14   0b11, 0x3C8
node       48   0x9F6BDECED846
-----------------------------------------
total      128
-----------------------------------------
final: C232AB00-9414-11EC-B3C8-9F6BDECED846
```

Figure 16: UUIDv1 Example Test Vector

A.2. Example of a UUIDv3 Value

The MD5 computation from is detailed in Figure 17 using the DNS Namespace ID value and the Name "www.example.com". The field mapping and all values are illustrated in Figure 18. Finally, to further illustrate the bit swapping for version and variant, see Figure 19.

```
Namespace (DNS):  6ba7b810-9dad-11d1-80b4-00c04fd430c8
Name:             www.example.com
-------------------------------------------------------
MD5:              5df418813aed051548a72f4a814cf09e
```

Figure 17: UUIDv3 Example MD5

```
-----------------------------------------
field     bits value
-----------------------------------------
md5_high  48   0x5df418813aed
ver        4   0x3
md5_mid   12   0x515
var        2   0b10
md5_low   62   0b00, 0x8a72f4a814cf09e
-----------------------------------------
total     128
-----------------------------------------
final: 5df41881-3aed-3515-88a7-2f4a814cf09e
```

Figure 18: UUIDv3 Example Test Vector

```
MD5 hex and dash:      5df41881-3aed-0515-48a7-2f4a814cf09e
Ver and Var Overwrite: xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx
Final:                 5df41881-3aed-3515-88a7-2f4a814cf09e
```

Figure 19: UUIDv3 Example Ver/Var Bit Swaps

A.3. Example of a UUIDv4 Value

This UUIDv4 example was created by generating 16 bytes of random data resulting in the hexadecimal value of 919108F752D133205BACF847DB4148A8. This is then used to fill out the fields as shown in Figure 20.

Finally, to further illustrate the bit swapping for version and variant, see Figure 21.

```
-----------------------------------------
field     bits value
-----------------------------------------
random_a  48   0x919108f752d1
ver        4   0x4
random_b  12   0x320
var        2   0b10
random_c  62   0b01, 0xbacf847db4148a8
-----------------------------------------
total     128
-----------------------------------------
final: 919108f7-52d1-4320-9bac-f847db4148a8
```

Figure 20: UUIDv4 Example Test Vector

```
Random hex:            919108f752d133205bacf847db4148a8
Random hex and dash:   919108f7-52d1-3320-5bac-f847db4148a8
Ver and Var Overwrite: xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx
Final:                 919108f7-52d1-4320-9bac-f847db4148a8
```

Figure 21: UUIDv4 Example Ver/Var Bit Swaps

A.4. Example of a UUIDv5 Value

The SHA-1 computation form is detailed in Figure 22, using the DNS Namespace ID value and the Name "www.example.com". The field mapping and all values are illustrated in Figure 23. Finally, to further illustrate the bit swapping for version and variant and the unused/discarded part of the SHA-1 value, see Figure 24.

```
Namespace (DNS):  6ba7b810-9dad-11d1-80b4-00c04fd430c8
Name:             www.example.com
---------------------------------------------------------
SHA-1:            2ed6657de927468b55e12665a8aea6a22dee3e35
```

Figure 22: UUIDv5 Example SHA-1

```
-----------------------------------------
field      bits value
-----------------------------------------
sha1_high  48   0x2ed6657de927
ver         4   0x5
sha1_mid   12   0x68b
var         2   0b10
sha1_low   62   0b01, 0x5e12665a8aea6a2
-----------------------------------------
total      128
-----------------------------------------
final: 2ed6657d-e927-568b-95e1-2665a8aea6a2
```

Figure 23: UUIDv5 Example Test Vector

```
SHA-1 hex and dash:    2ed6657d-e927-468b-55e1-2665a8aea6a2-2dee3e35
Ver and Var Overwrite: xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx
Final:                 2ed6657d-e927-568b-95e1-2665a8aea6a2
Discarded:                                                -2dee3e35
```

Figure 24: UUIDv5 Example Ver/Var Bit Swaps and Discarded SHA-1 Segment

A.5. Example of a UUIDv6 Value

```
-----------------------------------------
field       bits value
-----------------------------------------
time_high   32   0x1EC9414C
time_mid    16   0x232A
ver          4   0x6
time_high   12   0xB00
var          2   0b10
clock_seq   14   0b11, 0x3C8
node        48   0x9F6BDECED846
-----------------------------------------
total       128
-----------------------------------------
final: 1EC9414C-232A-6B00-B3C8-9F6BDECED846
```

Figure 25: UUIDv6 Example Test Vector

A.6. Example of a UUIDv7 Value

This example UUIDv7 test vector utilizes a well-known Unix Epoch timestamp with millisecond precision to fill the first 48 bits.

rand_a and rand_b are filled with random data.

The timestamp is Tuesday, February 22, 2022 2:22:22.00 PM GMT-05:00, represented as 0x017F22E279B0 or 1645557742000.

```
-----------------------------------------
field       bits value
-----------------------------------------
unix_ts_ms  48   0x017F22E279B0
ver          4   0x7
rand_a      12   0xCC3
var          2   0b10
rand_b      62   0b01, 0x8C4DC0C0C07398F
-----------------------------------------
total       128
-----------------------------------------
final: 017F22E2-79B0-7CC3-98C4-DC0C0C07398F
```

Figure 26: UUIDv7 Example Test Vector

Appendix B. Illustrative Examples

The following sections contain illustrative examples that serve to show how one may use UUIDv8 (Section 5.8) for custom and/or experimental application-based logic. The examples below have not been through the same rigorous testing, prototyping, and feedback loop that other algorithms in this document have undergone. The authors encourage implementers to create their own UUIDv8 algorithm rather than use the items defined in this section.

B.1. Example of a UUIDv8 Value (Time-Based)

This example UUIDv8 test vector utilizes a well-known 64-bit Unix Epoch timestamp with 10 ns precision, truncated to the least significant, rightmost bits to fill the first 60 bits of custom_a and custom_b, while setting the version bits between these two segments to the version value of 8.

The variant bits are set; and the final segment, custom_c, is filled with random data.

Timestamp is Tuesday, February 22, 2022 2:22:22.000000 PM GMT-05:00, represented as 0x2489E9AD2EE2E00 or 164555774200000000 (10 ns-steps).

```
-----------------------------------------
field     bits value
-----------------------------------------
custom_a  48   0x2489E9AD2EE2
ver        4   0x8
custom_b  12   0xE00
var        2   0b10
custom_c  62   0b00, 0xEC932D5F69181C0
-----------------------------------------
total     128
-----------------------------------------
final: 2489E9AD-2EE2-8E00-8EC9-32D5F69181C0
```

Figure 27: UUIDv8 Example Time-Based Illustrative Example

B.2. Example of a UUIDv8 Value (Name-Based)

As per Section 5.5, name-based UUIDs that want to use modern hashing algorithms **MUST** be created within the UUIDv8 space. These **MAY** leverage newer hashing algorithms such as SHA-256 or SHA-512 (as defined by [FIPS180-4]), SHA-3 or SHAKE (as defined by [FIPS202]), or even algorithms that have not been defined yet.

A SHA-256 version of the SHA-1 computation in Appendix A.4 is detailed in Figure 28 as an illustrative example detailing how this can be achieved. The creation of the name-based UUIDv8 value in this section follows the same logic defined in Section 5.5 with the difference being SHA-256 in place of SHA-1.

The field mapping and all values are illustrated in Figure 29. Finally, to further illustrate the bit swapping for version and variant and the unused/discarded part of the SHA-256 value, see Figure 30. An important note for secure hashing algorithms that produce outputs of an arbitrary size, such as those found in SHAKE, is that the output hash **MUST** be 128 bits or larger.

```
Namespace (DNS):       6ba7b810-9dad-11d1-80b4-00c04fd430c8
Name:                  www.example.com
----------------------------------------------------------------
SHA-256:
5c146b143c524afd938a375d0df1fbf6fe12a66b645f72f6158759387e51f3c8
```

Figure 28: UUIDv8 Example SHA256

```
-----------------------------------------
field     bits value
-----------------------------------------
custom_a  48   0x5c146b143c52
ver        4   0x8
custom_b  12   0xafd
var        2   0b10
custom_c  62   0b00, 0x38a375d0df1fbf6
-----------------------------------------
total     128
-----------------------------------------
final: 5c146b14-3c52-8afd-938a-375d0df1fbf6
```

Figure 29: UUIDv8 Example Name-Based SHA-256 Illustrative Example

```
A: 5c146b14-3c52-4afd-938a-375d0df1fbf6-fe12a66b645f72f6158759387e51f3c8
B: xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx
C: 5c146b14-3c52-8afd-938a-375d0df1fbf6
D:                                     -fe12a66b645f72f6158759387e51f3c8
```

Figure 30: UUIDv8 Example Ver/Var Bit Swaps and Discarded SHA-256 Segment

Examining Figure 30:

- Line A details the full SHA-256 as a hexadecimal value with the dashes inserted.
- Line B details the version and variant hexadecimal positions, which must be overwritten.
- Line C details the final value after the ver and var have been overwritten.
- Line D details the discarded leftover values from the original SHA-256 computation.

Acknowledgements

The authors gratefully acknowledge the contributions of Rich Salz, Michael Mealling, Ben Campbell, Ben Ramsey, Fabio Lima, Gonzalo Salgueiro, Martin Thomson, Murray S. Kucherawy, Rick van Rein, Rob Wilton, Sean Leonard, Theodore Y. Ts'o, Robert Kieffer, Sergey Prokhorenko, and LiosK.

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

This document draws heavily on the OSF DCE specification (Appendix A of [C309]) for UUIDs. Ted Ts'o provided helpful comments.

We are also grateful to the careful reading and bit-twiddling of Ralf S. Engelschall, John Larmouth, and Paul Thorpe. Professor Larmouth was also invaluable in achieving coordination with ISO/IEC.

Authors' Addresses

Kyzer R. Davis
Cisco Systems
Email: kydavis@cisco.com

Brad G. Peabody
Uncloud
Email: brad@peabody.io

P. Leach
University of Washington
Email: pjl7@uw.edu