rkgroup: dispatch
ternet-Draft:
aft-peabody-dispatch-new-uuid-format-04
dates: 4122 (if approved)
olished: 23 June 2022
tended Status: Standards Track
oires: 25 December 2022

BGP. Peabody

K. Davis

New UUID Formats

stract

This document presents new Universally Unique Identifier (UUID) formats for use in modern applications and databases.¶

atus of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 25 December 2022.

ble of Contents

Introduction

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 UUIDs have gained popularity is as database keys. This stems from the increasingly distributed nature of modern applications. In such cases, "auto increment" schemes often used by databases do not work well, as the effort required to coordinate unique 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 synchronization makes them a good alternative, but UUID versions 1-5 lack certain other desirable characteristics:

1. Non-time-ordered UUID versions such as UUIDv4 have poor database index locality. Meaning new values created in succession are not close to each other in the index and thus require inserts to be performed at random locations. The negative performance effects of which on common structures used for this (B-tree and its variants) can be dramatic.

2. The 100-nanosecond, Gregorian epoch used in UUIDv1 timestamps is uncommon and difficult to represent accurately using a standard number format such as [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 MAC address in the node field of Version 1 UUIDs. Exposed MAC addresses can be used as an attack surface to locate machines and reveal various other information about such machines (minimally manufacturer, potentially other details). Additionally, with the advent of virtual machines and containers, MAC address uniqueness is no longer guaranteed.

5. Many of the implementation details specified in [RFC4122] involve trade offs that are neither possible to specify for all applications nor necessary to produce interoperable implementations.

6. [RFC4122] does not distinguish between the requirements for generation of a UUID versus an application which simply stores one, which are often different.

Due to the aforementioned issue, 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 lead 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/encoding, timestamp type, timestamp format, timestamp accuracy, node format/components, collision handling and multi-timestamp tick generation sequencing.

1. [ULID] by A. Feerasta
2. [LexicalUUID] by Twitter
3. [Snowflake] by Twitter
4. [Flake] by Boundary
5. [ShardingID] by Instagram
6. [KSUID] by Segment
7. [Elasticflake] by P. Pearcy
8. [FlakeID] by T. Pawlak
9. [Sonyflake] by Sony
0. [orderedUuid] by IT. Cabrera
1. [COMBGUID] by R. Tallent
2. [SID] by A. Chilton
3. [pushID] by Google
4. [XID] by O. Poitrey
5. [ObjectID] by MongoDB
6. [CUID] by E. Elliott

An inspection of these implementations and the issues described above has led to this document which attempts to adapt UUIDs to address these issues.

Terminology

l. 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.

2. Abbreviations

The following abbreviations are used in this document:

UUID Universally Unique Identifier [RFC4122]

CSPRNG Cryptographically Secure Pseudo-Random Number Generator

MAC Media Access Control

MSB Most Significant Bit

DBMS Database Management System

Summary of Changes

The following UUIDs are hereby introduced:

UUID version 6 (UUIDv6)
A re-ordering of UUID version 1 so it is sortable as an opaque sequence of bytes. Easy to implement given an existing UUIDv1 implementation. See Section 5.1

UUID version 7 (UUIDv7)
An entirely new time-based UUID bit layout sourced from the widely implemented and well known Unix Epoch timestamp source. See Section 5.2

UUID version 8 (UUIDv8)
A free-form UUID format which has no explicit requirements except maintaining backward compatibility. See Section 5.3

Max UUID
A specialized UUID which is the inverse of [RFC4122], Section 4.1.7 See Section 5.4

1. changelog

RFC EDITOR PLEASE DELETE THIS SECTION.

draft-04

- Fixed bad title in IEEE754 Normative Reference
- Fixed bad GMT offset in Test Vector Appendix
- Removed MAY in Counters section
- Condensed Counter Type into Counter Methods to reduce text
- Removed option for random increment along with fixed-length counter
- Described how to handle scenario where New UUID less than Old UUID
- Allow timestamp increment if counter overflows
- Replaced UUIDv8 C code snippet with full generation example
- Fixed RFC4086 Reference link
- Describe reseeding best practice for CSPRNG

- Changed MUST to SHOULD removing requirement for absolute monotonicity

draft-03

- Reworked the draft body to make the content more concise
- UUIDv6 section reworked to just the reorder of the timestamp
- UUIDv7 changed to simplify timestamp mechanism to just millisecond Unix timestamp
- UUIDv8 relaxed to be custom in all elements except version and variant
- Introduced Max UUID.
- Added C code samples in Appendix.
- Added test vectors in Appendix.
- Version and Variant section combined into one section.
- Changed from pseudo-random number generators to cryptographically secure pseudo-random number generator (CSPRNG).
- Combined redundant topics from all UUIDs into sections such as Timestamp granularity, Monotonicity and Counters, Collision Resistance, Sorting, and Unguessability, etc.
- Split Encoding and Storage into Opacity and DBMS and Database Considerations
- Reworked Global Uniqueness under new section Global and Local Uniqueness
- Node verbiage only used in UUIDv6 all others reference random/rand instead
- Clock sequence verbiage changed simply to counter in any section other than UUIDv6
- Added Abbreviations section
- Updated IETF Draft XML Layout
- Added information about little-endian UUIDs

draft-02

- Added Changelog
- Fixed misc. grammatical errors
- Fixed section numbering issue
- Fixed some UUIDvX reference issues
- Changed all instances of "motonic" to "monotonic"
- Changed all instances of "#-bit" to "# bit"
- Changed "proceeding" verbiage to "after" in section 7
- Added details on how to pad 32 bit Unix timestamp to 36 bits in UUIDv7
- Added details on how to truncate 64 bit Unix timestamp to 36 bits in UUIDv7
- Added forward reference and bullet to UUIDv8 if truncating 64 bit Unix Epoch is not an option.
- Fixed bad reference to non-existent "time_or_node" in section 4.5.4

draft-01

- Complete rewrite of entire document.
- The format, flow and verbiage used in the specification has been reworked to mirror the original RFC 4122 and current IETF standards.
- Removed the topics of UUID length modification, alternate UUID text formats, and alternate UUID encoding techniques.
- Research into 16 different historical and current implementations of time-based universal identifiers was completed

at the end of 2020 in attempt to identify trends which have directly influenced design decisions in this draft document (https://github.com/uuid6/uuid6-ietf-draft/tree/master/research)
- Prototype implementation have been completed for UUIDv6, UUIDv7, and UUIDv8 in various languages by many GitHub community members. (https://github.com/uuid6/prototypes)

Variant and Version Fields

The variant bits utilized by UUIDs in this specification remain in the same octet as originally defined by [RFC4122], Section 4.1.1.

The next table details Variant 10xx (8/9/A/B) and the new versions defined by this specification. A complete guide to all versions within this variant has been includes in Appendix C.1.

sb0	Msb1	Msb2	Msb3	Version	Description
	1	1	0	6	Reordered Gregorian time-based UUID specified in this document.
	1	1	1	7	Unix Epoch time-based UUID specified in this document.
	0	0	0	8	Reserved for custom UUID formats specified in this document

Table 1: New UUID variant 10xx (8/9/A/B) versions defined by this specification

For UUID version 6, 7 and 8 the variant field placement from [RFC4122] are unchanged. An example version/variant layout for UUIDv6 follows the table where M is the version and N is the variant.

```
00000000-0000-6000-8000-000000000000
00000000-0000-6000-9000-000000000000
00000000-0000-6000-A000-000000000000
00000000-0000-6000-B000-000000000000
xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx
```

Figure 1: UUIDv6 Variant Examples

New Formats

The UUID format is 16 octets; the variant bits in conjunction with the version bits described in the next section in determine finer structure.

l. UUID Version 6

UUID version 6 is a field-compatible version of UUIDv1, reordered for improved DB locality. It is expected that UUIDv6 will primarily be used in contexts where there are existing v1 UUIDs. Systems that do not involve legacy UUIDv1 SHOULD consider using UUIDv7 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 [RFC4122], Section 4.1.4, 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 bits remain unchanged from their usage and position in [RFC4122], Section 4.1.5.

The 48 bit node SHOULD be set to a pseudo-random value however implementations MAY choose to retain the old MAC address behavior from [RFC4122], Section 4.1.6 and [RFC4122], Section 4.5. 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 1

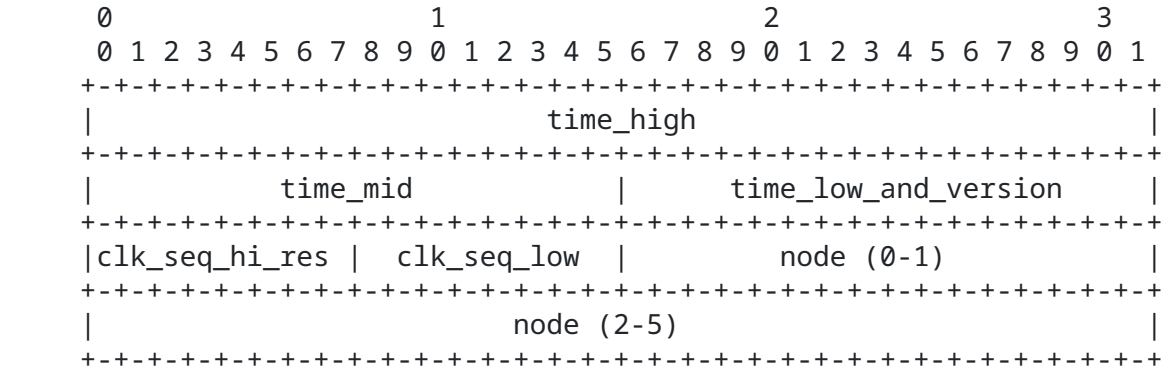

Figure 2: UUIDv6 Field and Bit Layout

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)

time_low_and_version:
The first four most significant bits MUST contain the UUIDv6 version (0110) while the remaining 12 bits will contain the least significant 12 bits from the 60 bit starting timestamp. Occupies bits 48 through 63 (octets 6-7)

clk_seq_hi_res:
The first two bits MUST be set to the UUID variant (10) The remaining 6 bits contain the high portion of the clock sequence. Occupies bits 64 through 71 (octet 8)

clock_seq_low:
The 8 bit low portion of the clock sequence. Occupies bits 72 through 79 (octet 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.

2. UUID Version 7

UUID version 7 features a time-ordered value field derived from the widely implemented and well known Unix Epoch timestamp source, the number of milliseconds seconds since midnight 1 Jan 1970 UTC, leap seconds excluded. As well as improved entropy characteristics over versions 1 or 6.

Implementations SHOULD utilize UUID version 7 over UUID version 1 and 6 if possible.

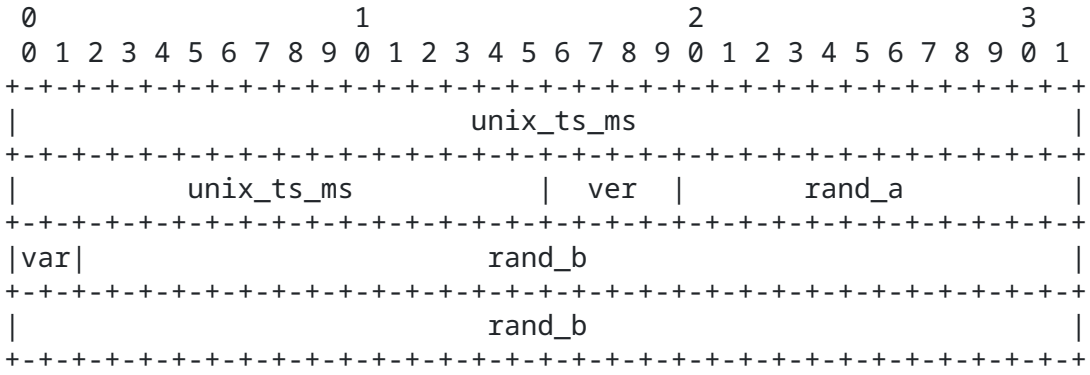

Figure 3: UUIDv7 Field and Bit Layout

unix_ts_ms:
48 bit big-endian unsigned number of Unix epoch timestamp as per Section 6.1.

ver:
4 bit UUIDv7 version set as per Section 4

rand_a:
12 bits pseudo-random data to provide uniqueness as per Section 6.2 and Section 6.6.

var:
The 2 bit variant defined by Section 4.

rand_b:
The final 62 bits of pseudo-random data to provide uniqueness as per Section 6.2 and Section 6.6.

3. UUID Version 8

UUID version 8 provides an RFC-compatible format for experimental or vendor-specific use cases. The only requirement is that the variant and version bits MUST be set as defined in Section 4. UUIDv8's uniqueness will be implementation-specific and SHOULD NOT be assumed.

The only explicitly defined bits are the Version and Variant leaving 122 bits for implementation specific time-based UUIDs. To be clear: UUIDv8 is not a replacement for UUIDv4 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/language restrictions
   which inhibit the use of one of the current UUIDs.

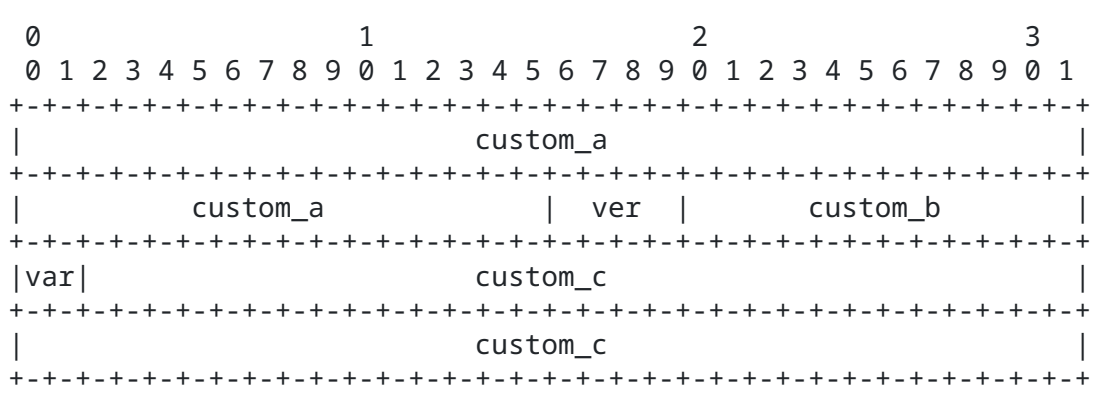

Figure 4: UUIDv8 Field and Bit Layout

custom_a:
   The first 48 bits of the layout that can be filled as an
   implementation sees fit.

ver:
   The 4 bit version field as defined by Section 4

custom_b:
   12 more bits of the layout that can be filled as an implementation
   sees fit.

var:
   The 2 bit variant field as defined by Section 4.

custom_c:
   The final 62 bits of the layout immediatly following the var field
   to be filled as an implementation sees fit.

4. Max UUID

The Max UUID is 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 Nil
UUID defined in [RFC4122], Section 4.1.7

FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF

Figure 5: Max UUID Format

UUID Best Practices

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

1. Timestamp Granularity

UUID timestamp source, precision and length was the topic of great
debate while creating this specification. As such choosing the right
timestamp for your application is a very important topic. This
section will detail some of the most common points on this topic.

Reliability:
Implementations SHOULD use the current timestamp from a reliable source to provide values that are time-ordered and continually increasing. Care SHOULD 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 must decide how to handle such cases. (See Altering, Fuzzing, or Smearing bullet below.)

Source:
UUID version 1 and 6 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 SHOULD be leveraged.

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 and it is frequently less than the precision offered by the operating system. With UUID version 1 and 6, 100-nanoseconds of precision are present while UUIDv7 features fixed millisecond level of precision within the Unix epoch that does not exceed the granularity capable in most modern systems. For other levels of precision UUIDv8 SHOULD be utilized.

Length:
The length of a given timestamp directly impacts how long a given UUID will be valid. That is, how many timestamp ticks can be contained in a UUID before the maximum value for the timestamp field is reached. Care should be given to ensure that the proper length is selected for a given timestamp. UUID version 1 and 6 utilize a 60 bit timestamp and UUIDv7 features a 48 bit timestamp.

Altering, Fuzzing, or Smearing:
Implementations MAY alter the actual timestamp. Some examples included security considerations around providing a real clock value within a UUID, to correct inaccurate clocks or to handle leap seconds. This specification makes no requirement or guarantee about how close the clock value needs to be to actual time.

Padding:
When timestamp padding is required, implementations MUST pad the most significant bits (left-most) bits with zeros. An example is padding the most significant, left-most bits of a 32 bit Unix timestamp with zero's to fill out the 48 bit timestamp in UUIDv7.

Truncating:
Similarly, 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, right-most 48 bits for UUIDv7.

2. Monotonicity and Counters

Monotonicity is the backbone of time-based sortable UUIDs. Naturally

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.

Additionally, care SHOULD be taken to ensure UUIDs generated in batches are also monotonic. That is, if one-thousand UUIDs are generated for the same timestamp; there is sufficient logic for organizing the creation order of those one-thousand UUIDs. For batch UUID creation implementions MAY utilize a monotonic counter which SHOULD increment 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 UUID version 1, 6, and 7 can provide sufficient monotonicity guarantees by simply ensuring that timestamp increments before creating a new UUID. For the topic of Distributed Nodes please refer to Section 6.3

Implementations SHOULD choose one method for single-node UUID implementations that require batch UUID creation.

Fixed-Length Dedicated Counter Bits (Method 1):
This references the practice of allocating 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. Positioning of a fixed bit-length counter SHOULD be immediatly after the embedded timestamp. This promotes sortability and allows random data generation for each counter increment. With this method rand_a section of UUIDv7 SHOULD be utilized as fixed-length dedicated counter bits that are incremented by one for every UUID generation. The trailing random bits generated for each new UUID in rand_b can help produce unguessable UUIDs. In the event more counter bits are required the most significant, left-most, bits of rand_b MAY be leveraged as additional counter bits.

Monotonic Random (Method 2):
With this method the random data is extended to also double as a counter. This monotonic random can be thought of as a "randomly seeded counter" which 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 SHOULD be a random integer of any desired length larger than zero. It ensures the UUIDs retain the required level of unguessability characters provided by the underlying entropy. The increment value MAY be one when the amount of UUIDs generated in a particular period of time is important and guessability is not an issue. However, it SHOULD NOT be used by implementations that favor unguessiblity, as the resulting values are easily guessable.

The following sub-topics cover topics related solely with creating reliable fixed-length dedicated counters:

Fixed-Length Dedicated Counter Seeding:
Implementations utilizing fixed-length counter method SHOULD randomly initialize the counter with each new timestamp tick. However, when the timestamp has not incremented; the counter SHOULD be frozen and incremented via the desired increment logic. When utilizing a randomly seeded counter alongside Method 1; the random 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-length counter method MAY also choose to randomly initialize a portion counter rather than the entire counter. For example, a 24 bit counter could have the 23 bits in least-significant, right-most, position randomly initialized. The remaining most significant, left-most counter bits are initialized as zero for the sole purpose of guarding against counter rollovers.

Fixed-Length Dedicated Counter Length:
Care MUST be taken to select a counter bit-length that can properly handle the level of timestamp precision in use. For example, millisecond precision SHOULD require 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 SHOULD also be given 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-Length Dedicated Counter Seeding which describes allocating a segment of the fixed-length counter as a rollover guard is also helpful to mitigate counter rollover issues. This same technique can be leveraged with Monotonic random counter methods by ensuring the total length of a possible increment in the least significant, right most position is less than the total length of the random being incremented. As such the most significant, left-most, bits can be incremented as rollover guarding.

Counter Rollover Handling:
Counter rollovers SHOULD 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).

Implementations SHOULD check if the 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, but not

limited to, clock rollbacks, leap second handling or counter rollovers. Applications SHOULD embed sufficient logic to catch these scenarios and correct the problem ensuring the next UUID generated is greater than the previous. To handle this scenario, the general guidance is that application MAY reuse the previous timestamp and increment the previous counter method.

3. Distributed UUID Generation

Some implementations MAY desire to utilize multi-node, clustered, applications which 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 nodes increase; so does the likelihood of a collision. This section will detail the approaches that MAY be utilized by multi-node UUID implementations in distributed environments.

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. Utilization of shared knowledge schemes with central/global registries is outside the scope of this specification.

Node IDs:
With this method, a pseudo-random 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 as 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.

Utilization of either a Centralized Registry or Node ID are not required for implementing UUIDs in this specification. However implementations SHOULD utilize one of the two aforementioned methods if distributed UUID generation is a requirement.

4. Collision Resistance

Implementations SHOULD weigh the consequences of UUID collisions within their application and when deciding between UUID versions that use entropy (random) versus the other components such as Section 6.1 and Section 6.2. This is especially true for distributed node collision resistance as defined by Section 6.3.

There are two example scenarios below which 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 the traditional UUID format.

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 and failure is unacceptable. Applications dealing with this type of scenario MUST employ as much collision resistance as possible within the given application context.

5. 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 UUID to provide uniqueness guarantees. Implementations MAY implement a shared knowledge scheme introduced in Section 6.3 as they see fit to extend the uniqueness guaranteed this specification and [RFC4122].

5. Unguessability

Implementations SHOULD utilize a cryptographically secure pseudo-random number generator (CSPRNG) to provide values that are both difficult to predict ("unguessable") and have a low likelihood of collision ("unique"). Care SHOULD be taken 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 Section 6.4 and Section 8 are present in modern UUIDs.

Advice on generating cryptographic-quality random numbers can be found in [RFC4086]

7. Sorting

UUIDv6 and UUIDv7 are designed so that implementations that require sorting (e.g. database indexes) SHOULD sort as opaque raw bytes, without 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 vs random data inserts can be quite large.

UUIDs formats created by this specification SHOULD be Lexicographically sortable while in the textual representation.

UUIDs created by this specification are crafted with big-ending byte order (network byte order) in mind. If Little-endian style is required a custom UUID format SHOULD be created using UUIDv8.

3. Opacity

UUIDs SHOULD be treated as opaque values and implementations SHOULD NOT examine the bits in a UUID to whatever extent is possible. However, where necessary, inspectors should refer to Section 4 for

more information on determining UUID version and variant.

9. 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 such:

* Storing as binary 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 and thus maybe 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-generate UUIDs) provides the best UUID monotonicity. In addition to UUIDs, additional identifiers MAY be used to ensure integrity and feedback.

IANA Considerations

This document has no IANA actions.

Security Considerations

MAC addresses pose inherent security risks 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 Section 6.6 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 it's 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 [RFC4122] UUIDv4 SHOULD be utilized.

Acknowledgements

The authors gratefully acknowledge the contributions of 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, sergeyprokhorenko, LiosK As well as all of those in the IETF community and on GitHub to who contributed to the discussions which resulted in this document.

Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate

Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[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>.

[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>.

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

. Informative References

[LexicalUUID] Twitter, "A Scala client for Cassandra", commit f6da4e0, November 2012, <https://github.com/twitter-archive/cassie>.

[Snowflake] Twitter, "Snowflake is a network service for generating unique ID numbers at high scale with some simple guarantees.", Commit b3f6a3c, May 2014, <https://github.com/twitter-archive/snowflake/releases/tag/snowflake-2010>.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ɔendix A. Example Code

l. Creating a UUIDv6 Value

This section details a function in C which converts from a UUID version 1 to version 6:

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include <arpa/inet.h>
#include <uuid/uuid.h>

/* Converts UUID version 1 to version 6 in place. */
void uuidv1tov6(uuid_t u) {

  uint64_t ut;
  unsigned char *up = (unsigned char *)u;

  // load ut with the first 64 bits of the UUID
  ut = ((uint64_t)ntohl(*((uint32_t*)up))) << 32;
  ut |= ((uint64_t)ntohl(*((uint32_t*)&up[4])));

  // dance the bit-shift...
  ut =
    ((ut >> 32) & 0x0FFF) | // 12 least significant bits
    (0x6000) | // version number
    ((ut >> 28) & 0x0000000FFFFF0000) | // next 20 bits
    ((ut << 20) & 0x000FFFF000000000) | // next 16 bits
    (ut << 52); // 12 most significant bits

  // store back in UUID
  *((uint32_t*)up) = htonl((uint32_t)(ut >> 32));
  *((uint32_t*)&up[4]) = htonl((uint32_t)(ut));

}
```

Figure 6: UUIDv6 Function in C

2. Creating a UUIDv7 Value

```
nclude <stdio.h>
nclude <stdlib.h>
nclude <stdint.h>
nclude <string.h>
nclude <time.h>

 ...

 csprng data source
_E *rndf;
df = fopen("/dev/urandom", "r");
 (rndf == 0) {
  printf("fopen /dev/urandom error\n");
  return 1;


 ...

 generate one UUIDv7E
nt8_t u[16];
truct timespec ts;
t ret;

t = clock_gettime(CLOCK_REALTIME, &ts);
 (ret != 0) {
  printf("clock_gettime error: %d\n", ret);
  return 1;


nt64_t tms;

s = ((uint64_t)ts.tv_sec) * 1000;
s += ((uint64_t)ts.tv_nsec) / 1000000;

nset(u, 0, 16);

ead(&u[6], 10, 1, rndf); // fill everything after the timestamp with r

(uint64_t*)(u)) |= htonll(tms << 16); // shift time into first 48 bits

8] = 0x80 | (u[8] & 0x3F); // set variant field, top two bits are 1, 0
6] = 0x70 | (u[6] & 0x0F); // set version field, top four bits are 0,
```

Figure 7: UUIDv7 Function in C

3. Creating a UUIDv8 Value

UUIDv8 will vary greatly from implementation to implementation.

The following example utilizes:

* 32 bit custom-epoch timestamp (seconds elapsed since 2020-01-01 00:00:00 UTC)

* 16 bit exotic resolution (~15 microsecond) subsecond timestamp encoded using the fractional representation

* 58 bit random number

* 8 bit application-specific unique node ID

```
 *  8 bit rolling sequence number

nclude <stdint.h>
nclude <time.h>

t get_random_bytes(uint8_t *buffer, int count) {
// ...


t generate_uuidv8(uint8_t *uuid, uint8_t node_id) {
struct timespec tp;
if (clock_gettime(CLOCK_REALTIME, &tp) != 0)
  return -1; // real-time clock error

// 32 bit biased timestamp (seconds elapsed since 2020-01-01 00:00:00
uint32_t timestamp_sec = tp.tv_sec - 1577836800;
uuid[0] = timestamp_sec >> 24;
uuid[1] = timestamp_sec >> 16;
uuid[2] = timestamp_sec >> 8;
uuid[3] = timestamp_sec;

// 16 bit subsecond fraction (~15 microsecond resolution)
uint16_t timestamp_subsec = ((uint64_t)tp.tv_nsec << 16) / 1000000000;
uuid[4] = timestamp_subsec >> 8;
uuid[5] = timestamp_subsec;

// 58 bit random number and required ver and var fields
if (get_random_bytes(&uuid[6], 8) != 0)
  return -1; // random number generator error
uuid[6] = 0x80 | (uuid[6] & 0x0f);
uuid[8] = 0x80 | (uuid[8] & 0x3f);

// 8 bit application-specific node ID to guarantee application-wide un
uuid[14] = node_id;

// 8 bit rolling sequence number to help ensure process-wide uniquenes
static uint8_t sequence = 0;
uuid[15] = sequence++; // NOTE: unprotected from race conditions

return 0;
```

Figure 8: UUIDv8 Function in C

pendix B. 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 clk_seq_hi_res, clock_seq_low, and node. All of which have been generated with random data.

```
Unix Nanosecond precision to Gregorian 100-nanosecond intervals
egorian_100_ns = (Unix_64_bit_nanoseconds / 100) + gregorian_Unix_offs

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.
gregorian_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_nanoseconds = 0x16D6320C3D4DCC00 or 1645557742000000000

Work:
gregorian_100_ns = (1645557742000000000 / 100) + 122192928000000000
(138648505420000000 - 122192928000000000) * 100 = Unix_64_bit_nanoseco

Final:
gregorian_100_ns = 0x1EC9414C232AB00 or 138648505420000000

Original: 00011110110010010100000101001100001000110010101010110000000000
UUIDv1:   11000010001100101010101100000000|1001010000010100|0001|00011
UUIDv6:   00011110110010010100000101001100|0010001100101010|0110|10110
```

Figure 9: Test Vector Timestamp Pseudo-code

4. Example of a UUIDv6 Value

field	bits	value_hex
time_low	32	0xC232AB00
time_mid	16	0x9414
time_hi_and_version	16	0x11EC
clk_seq_hi_res	8	0xB3
clock_seq_low	8	0xC8
node	48	0x9E6BDECED846
total	128	

final_hex: C232AB00-9414-11EC-B3C8-9E6BDECED846

Figure 10: UUIDv1 Example Test Vector

field	bits	value_hex
time_high	32	0x1EC9414C
time_mid	16	0x232A
time_low_and_version	16	0x6B00
clk_seq_hi_res	8	0xB3
clock_seq_low	8	0xC8
node	48	0x9E6BDECED846
total	128	

final_hex: 1EC9414C-232A-6B00-B3C8-9E6BDECED846

Figure 11: UUIDv6 Example Test Vector

5. Example of a UUIDv7 Value

```
This example UUIDv7 test vector utilizes a well-known 32 bit Unix
epoch with additional 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 0x17F22E279B0 or 1645557742000

-------------------------------
field      bits    value
-------------------------------
unix_ts_ms   48    0x17F22E279B0
var           4    0x7
rand_a       12    0xCC3
var           2    b10
rand_b       62    0x18C4DC0C0C07398F
-------------------------------
total       128
-------------------------------
final: 017F22E2-79B0-7CC3-98C4-DC0C0C07398F
```

Figure 12: UUIDv7 Example Test Vector

B. Example of a UUIDv8 Value

```
This example UUIDv8 test vector utilizes a well-known 64 bit Unix
epoch with nanosecond precision, truncated to the least-significant,
right-most, bits to fill the first 48 bits through version.

The next two segments of custom_b and custom_c are are filled with
random data.

Timestamp is Tuesday, February 22, 2022 2:22:22.000000 PM GMT-05:00
represented as 0x16D6320C3D4DCC00 or 1645557742000000000

It should be noted that this example is just to illustrate one
scenario for UUIDv8. Test vectors will likely be implementation
specific and vary greatly from this simple example.

-------------------------------
field      bits    value
-------------------------------
custom_a     48    0x320C3D4DCC00
ver           4    0x8
custom_b     12    0x75B
var           2    b10
custom_c     62    0xEC932D5F69181C0
-------------------------------
total       128
-------------------------------
final: 320C3D4D-CC00-875B-8EC9-32D5F69181C0
```

Figure 13: UUIDv8 Example Test Vector

Appendix C. Version and Variant Tables

C.1. Variant 10xx Versions

Msb0	Msb1	Msb2	Msb3	Version	Description

	0	0	0	0	Unused
	0	0	1	1	The Gregorian time-based UUID from in [RFC4122], Section 4.1.3
	0	1	0	2	DCE Security version, with embedded POSIX UIDs from [RFC4122], Section 4.1.3
	0	1	1	3	The name-based version specified in [RFC4122], Section 4.1.3 that uses MD5 hashing.
	1	0	0	4	The randomly or pseudo-randomly generated version specified in [RFC4122], Section 4.1.3.
	1	0	1	5	The name-based version specified in [RFC4122], Section 4.1.3 that uses SHA-1 hashing.
	1	1	0	6	Reordered Gregorian time-based UUID specified in this document.
	1	1	1	7	Unix Epoch time-based UUID specified in this document.
	0	0	0	8	Reserved for custom UUID formats specified in this document.
	0	0	1	9	Reserved for future definition.
	0	1	0	10	Reserved for future definition.
	0	1	1	11	Reserved for future definition.
	1	0	0	12	Reserved for future definition.
	1	0	1	13	Reserved for future definition.
	1	1	0	14	Reserved for future definition.
	1	1	1	15	Reserved for future definition.

Table 2: All UUID variant 10xx (8/9/A/B) version definitions.

thors' Addresses

Brad G. Peabody

Email: brad@peabody.io

Kyzer R. Davis

Email: kydavis@cisco.com