International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

X.667
(10/2012)

SERIES X: DATA NETWORKS, OPEN SYSTEM COMMUNICATIONS AND SECURITY

OSI networking and system aspects – Naming, Addressing and Registration

Information technology – Procedures for the operation of object identifier registration authorities: Generation of universally unique identifiers and their use in object identifiers

Recommendation ITU-T X.667

ITU-T X-SERIES RECOMMENDATIONS

DATA NETWORKS, OPEN SYSTEM COMMUNICATIONS AND SECURITY

PUBLIC DATA NETWORKS	
Services and facilities	X.1–X.19
Interfaces	X.20–X.49
Transmission, signalling and switching	X.50–X.89
Network aspects	X.90–X.149
Maintenance	X.150–X.179
Administrative arrangements	X.180–X.199
OPEN SYSTEMS INTERCONNECTION	
Model and notation	X.200–X.209
Service definitions	X.210–X.219
Connection-mode protocol specifications	X.220–X.229
Connectionless-mode protocol specifications	X.230–X.239
PICS proformas	X.240–X.259
Protocol Identification	X.260–X.269
Security Protocols	X.270–X.279
Layer Managed Objects	X.280–X.289
Conformance testing	X.290–X.299
INTERWORKING BETWEEN NETWORKS	
General	X.300–X.349
Satellite data transmission systems	X.350–X.369
IP-based networks	X.370–X.379
MESSAGE HANDLING SYSTEMS	X.400–X.499
DIRECTORY	X.500–X.599
OSI NETWORKING AND SYSTEM ASPECTS	
Networking	X.600–X.629
Efficiency	X.630–X.639
Quality of service	X.640–X.649
Naming, Addressing and Registration	**X.650–X.679**
Abstract Syntax Notation One (ASN.1)	X.680–X.699
OSI MANAGEMENT	
Systems management framework and architecture	X.700–X.709
Management communication service and protocol	X.710–X.719
Structure of management information	X.720–X.729
Management functions and ODMA functions	X.730–X.799
SECURITY	X.800–X.849
OSI APPLICATIONS	
Commitment, concurrency and recovery	X.850–X.859
Transaction processing	X.860–X.879
Remote operations	X.880–X.889
Generic applications of ASN.1	X.890–X.899
OPEN DISTRIBUTED PROCESSING	X.900–X.999
INFORMATION AND NETWORK SECURITY	X.1000–X.1099
SECURE APPLICATIONS AND SERVICES	X.1100–X.1199
CYBERSPACE SECURITY	X.1200–X.1299
SECURE APPLICATIONS AND SERVICES	X.1300–X.1399
CYBERSECURITY INFORMATION EXCHANGE	X.1500–X.1599

For further details, please refer to the list of ITU-T Recommendations.

INTERNATIONAL STANDARD ISO/IEC 9834-8
RECOMMENDATION ITU-T X.667

Information technology – Procedures for the operation of object identifier registration authorities: Generation of universally unique identifiers and their use in object identifiers

Summary

Recommendation ITU-T X.667 | ISO/IEC 9834-8 specifies procedures for the generation of universally unique identifiers (UUIDs) and for their use in the international object identifier tree under the joint UUID arc.

History

Edition	Recommendation	Approval	Study Group	Unique ID[*]
1.0	ITU-T X.667	2004-09-13	17	11.1002/1000/7290
2.0	ITU-T X.667	2008-08-29	17	11.1002/1000/9445
3.0	ITU-T X.667	2012-10-14	17	11.1002/1000/11746

[*] To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, http://handle.itu.int/11.1002/1000/11830-en.

CONTENTS

Page

Introduction

This Recommendation | International Standard standardizes the generation of universally unique identifiers (UUIDs).

UUIDs are an octet string of 16 octets (128 bits). The 16 octets can be interpreted as an unsigned integer encoding, and the resulting integer value can be used as the primary integer value (defining an integer-valued Unicode label) for an arc of the International Object Identifier tree under the Joint UUID arc. This enables users to generate object identifier and OID internationalized resource identifier names without any registration procedure.

UUIDs are also known as globally unique identifiers (GUIDs), but this term is not used in this Recommendation | International Standard. UUIDs were originally used in the network computing system (NCS) [7] and later in the Open Software Foundation's Distributed Computing Environment (DCE) [6]. ISO/IEC 11578 [5] contains a short definition of some (but not all) of the UUID formats specified in this Recommendation | International Standard. The specification in this Recommendation | International Standard is consistent with all these earlier specifications.

UUIDs forming a component of an OID are represented in ASN.1 value notation as the decimal representation of their integer value, but for all other display purposes it is more usual to represent them with hexadecimal digits with a hyphen separating the different fields within the 16-octet UUID. This representation is defined in this Recommendation | International Standard.

If generated according to one of the mechanisms defined in this Recommendation | International Standard, a UUID is either guaranteed to be different from all other UUIDs generated before 3603 A.D., or is extremely likely to be different (depending on the mechanism chosen).

No centralized authority is required to administer UUIDs. Centrally generated UUIDs are guaranteed to be different from all other UUIDs centrally generated.

A UUID can be used for multiple purposes, from tagging objects with an extremely short lifetime, to reliably identifying very persistent objects across a network, particularly (but not necessarily) as part of an object identifier or OID internationalized resource identifier value, or in a uniform resource name (URN).

The UUID generation algorithm specified in this Recommendation | International Standard supports very high allocation rates: 10 million per second per machine if necessary, so UUIDs can also be used as transaction IDs. An informative annex provides a program in the C language that will generate UUIDs in accordance with this Recommendation | International Standard.

Three algorithms are specified for the generation of unique UUIDs, using different mechanisms to ensure uniqueness. These produce different versions of a UUID.

The first (and most common) mechanism produces the so-called time-based version. These UUIDs can be generated at the rate of 10 million per second. For UUIDs generated within a single computer system, a 60-bit time-stamp (used as a Clock value) with a granularity of 100 nanoseconds, based on coordinated universal time (UTC) is used to guarantee uniqueness over a period of approximately 1600 years. For UUIDs generated with the same time-stamp by different systems, uniqueness is obtained by use of 48-bit media access control (MAC) addresses, specified in ISO/IEC 8802-3 (this is used as a Node value). (These addresses are usually already available on most networked systems, but are otherwise obtainable from the IEEE Registration Authority for MAC addresses – see [4].) Alternative ways of generating Clock and Node values are specified for the time-based version if UTC time is not available on a system, or if there is no MAC address available.

The second mechanism produces a single UUID that is a name-based version, where cryptographic hashing is used to produce the 128-bit UUID value from a globally unambiguous (text) name.

The third mechanism uses pseudo-random or truly random number generation to produce most of the bits in the 128-bit value.

Clause 5 specifies the notation used for octet-order and bit-order naming, and for specification of transmission order.

Clause 6 specifies the structure of a UUID and the representation of it in binary, hexadecimal, or as a single integer value.

Clauses 7 and 8 specify the use of a UUID in an OID or a URN respectively.

Clause 9 specifies rules for comparing UUIDs to test for equality or to provide an ordering relation between two UUIDs.

Clause 10 discusses the possibility of checking the validity of a UUID. In general, UUIDs have little redundancy, and there is little scope for checking their validity.

Clause 11 describes the historical use of some bits in the UUID to define different variants of the UUID format, and specifies the value of these bits for UUIDs defined in accordance with this Recommendation | International Standard.

Clause 12 specifies the use of the fields of a UUID in the different versions that are defined (time-based, name-based, and random-number based versions). It also defines the transmission byte order.

Clause 13 specifies the setting of the fields of a time-based UUID.

Clause 14 specifies the setting of the fields of a name-based UUID.

Clause 15 specifies the setting of the fields of a random-number-based UUID.

All annexes are informative.

Annex A describes various algorithms for the efficient generation of time-based UUIDs.

Annex B discusses the properties that a name-based UUID should have, affecting the selection of name spaces for use in generating such UUIDs.

Annex C provides guidance on mechanisms that can be used to generate random numbers in a computer system.

Annex D contains a complete program in the C programming language that can be used to generate UUIDs.

INTERNATIONAL STANDARD
RECOMMENDATION ITU-T

Information technology – Procedures for the operation of object identifier registration authorities: Generation of universally unique identifiers and their use in object identifiers

1 Scope

This Recommendation | International Standard specifies the format and generation rules that enable users to produce 128-bit identifiers that are either guaranteed to be globally unique, or are globally unique with a high probability.

The universally unique identifiers (UUIDs) generated in conformance with this Recommendation | International Standard are suitable either for transient use, with generation of a new UUID every 100 nanoseconds, or as persistent identifiers.

This Recommendation | International Standard is derived from earlier non-standard specifications of UUIDs and their generation, and is technically identical to those earlier specifications.

This Recommendation | International Standard also specifies and allows the use of UUIDs as primary values (which define Unicode labels) for arcs beneath the Joint UUID arc. This enables users to generate and use such arcs without any registration procedures.

This Recommendation | International Standard also specifies and allows the use of UUIDs to form a uniform resource name (URN).

2 Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

2.1 Identical Recommendations | International Standards

– Recommendation ITU-T X.660 (2011) | ISO/IEC 9834-1:2012, *Information technology – Procedures for the operation of object identifier registration authorities: General procedures and top arcs of the international object identifier tree*.

– Recommendation ITU-T X.680 (2008) | ISO/IEC 8824-1:2008, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*.

2.2 Other normative references

– ISO/IEC 8802-3:2000, *Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*.

– ISO/IEC 10118-3:2004, *Information technology – Security techniques – Hash functions – Part 3: Dedicated hash-functions*.

– ISO/IEC 10646:2012, *Information technology – Universal Coded Character Set (UCS).*

– IETF RFC 1321 (1992), *The MD5 Message-Digest Algorithm*.

– IETF RFC 2141 (1997), *URN Syntax.*

– FIPS PUB 180-3:2008, *Federal Information Processing Standards Publication, Secure Hash Standard (SHS).*

3 Terms and definitions

For the purposes of this Recommendation | International Standard, the following definitions apply.

3.1 ASN.1 notation

This Recommendation | International Standard uses the following terms defined in Rec. ITU-T X.680 | ISO/IEC 8824-1:

a) Coordinated universal time (UTC);

b) object identifier type;

c) OID internationalized resource identifier type.

3.2 Registration authorities

This Recommendation | International Standard uses the following terms defined in Rec. ITU-T X.660 | ISO/IEC 9834-1:

a) International Object Identifier tree;

b) IRI/URI value;

c) OID internationalized resource identifier;

d) object identifier;

e) registration;[2]

f) registration authority;

g) registration procedures;

h) secondary identifier;

i) Unicode character;

j) Unicode label.

3.3 Network terms

This Recommendation | International Standard uses the following term defined in ISO/IEC 8802-3:

– MAC address.

3.4 Additional definitions

3.4.1 cryptographic-quality random-number: A random number or pseudo-random number generated by a mechanism, which ensures sufficient spread of repeatedly-generated values to be acceptable for use in cryptographic work (and is used in such work).

3.4.2 Joint UUID arc: An arc beneath the node of the International Object Identifier tree identified by the ASN.1 object identifier value `{joint-iso-itu-t(2) uuid(25)}` and the ASN.1 OID internationalized resource identifier value `"/UUID"`.

3.4.3 name-based version: A UUID that is generated using cryptographic hashing of a name space name and a name in that name space.

3.4.4 name space: A system for generating names of objects that ensures unambiguous identification within that name space.

NOTE – Examples of name spaces are the network domain name system, URNs, OIDs, Directory distinguished names (see [1]), and reserved words in a programming language.

3.4.5 random-number-based version: A UUID that is generated using a random or pseudo-random number.

3.4.6 standard UUID variant: The variant of the possible UUID formats that is specified by this Recommendation | International Standard.

NOTE – Historically, there have been other specifications of UUID formats that differ from the variant specified in this Recommendation | International Standard. UUIDs generated according to all these variant formats are all distinct.

3.4.7 time-based version: A UUID in which uniqueness is obtained by the use of a MAC address to identify a system, and a Clock value based on the current UTC time.

3.4.8 universally unique identifier (UUID): A 128-bit value generated in accordance with this Recommendation | International Standard, or in accordance with some historical specifications, and providing unique values between systems and over time (see also 3.4.6).

4 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
GUID	Globally Unique Identifier
MAC	Media Access Control
MD5	Message Digest algorithm 5
OID	Object Identifier
OID-IRI	OID internationalized resource identifier
SHA-1	Secure Hash Algorithm 1
URL	Uniform Resource Locator
URN	Uniform Resource Name
UTC	Coordinated Universal Time
UUID	Universally Unique Identifier

5 Notation

5.1 This Recommendation | International Standard specifies a sequence of octets for a UUID using the terms first and last. The first octet is also called "octet 15" and the last octet "octet 0".

5.2 The bits within a UUID are also numbered as "bit 127" to "bit 0", with bit 127 as the most significant bit of octet 15 and bit 0 as the least significant bit of octet 0.

5.3 When figures and tables are used in this Recommendation | International Standard, the most significant octet (and the most significant bit) are displayed on the left of the page. This corresponds with a transmission order of octets in which the left-most octets are transmitted first.

5.4 A number of values used in this Specification are expressed as the value of an unsigned integer of a given bit-length (N say). The bits of the N-bit unsigned integer value are numbered "bit N-1" to "bit 0", with bit N-1 as the most significant bit and bit 0 as the least significant bit.

5.5 These notations are used solely for the purposes of this Specification. Representations in computer memory are not standardized, and depend on the system architecture.

6 UUID structure and representation

6.1 UUID field structure

6.1.1 A UUID is specified as an ordered sequence of six fields. A UUID is specified in terms of the concatenation of these UUID fields. The UUID fields are named:

a) the "TimeLow" field;

b) the "TimeMid" field;

c) the "VersionAndTimeHigh" field;

d) the "VariantAndClockSeqHigh" field;

e) the "ClockSeqLow" field;

f) the "Node" field.

6.1.2 The UUID fields are defined to have a significance in the order listed above, with "TimeLow" as the most significant field (bit 31 of "TimeLow" is bit 127 of the UUID), and "Node" as the least significant field (bit 0 of "Node" is bit 0 of the UUID).

6.1.3 The contents of these UUID fields are specified in terms of a Version, Variant, Time, Clock Sequence, and Node unsigned integer value (each with a fixed bit-size). The setting of these values is specified in clause 12 and their mapping to the above UUID fields is specified in 12.1.

NOTE – As part of the names of some of the UUID fields (for example, TimeLow, TimeMid, and TimeHigh) imply, the sequential order of the bits in a UUID (bit 127 to bit 0) that derive from a particular unsigned integer value (for example, from bits 59 to 0 of the Time value) is not the same as the sequential order of the bits in that unsigned integer value. This is for historical reasons.

6.2 Binary representation

6.2.1 A UUID shall be represented in binary as 16 octets formed by the concatenation of the unsigned integer fixed-length encoding of each of its fields into one or more octets. The number of octets to be used for each field shall be:

a) the "TimeLow" field: four octets;

b) the "TimeMid" field: two octets;

c) the "VersionAndTimeHigh" field: two octets;

d) the "VariantAndClockSeqHigh" field: one octet;

e) the "ClockSeqLow" field: one octet;

f) the "Node" field: six octets.

NOTE – This order of UUID fields is the usual representation within a computer system, and in the hexadecimal text representation (see 6.4).

6.2.2 The most significant bit of the unsigned integer encoding of each UUID field shall be the most significant bit of its first octet (octet N, the most significant octet), and the least significant bit of the unsigned integer encoding shall be the least significant bit of its last octet (octet 0, the least significant bit).

6.2.3 The UUID fields shall be concatenated in the order of their significance (see 6.1.2) with the most significant field first and the least significant field last.

6.3 Representation as a single integer value

A UUID can be represented as a single integer value. To obtain the single integer value of the UUID, the 16 octets of the binary representation shall be treated as an unsigned integer encoding with the most significant bit of the integer encoding as the most significant bit (bit 7) of the first of the sixteen octets (octet 15) and the least significant bit as the least significant bit (bit 0) of the last of the sixteen octets (octet 0).

NOTE – The single integer value is used when the UUID forms the primary integer value of a Joint UUID arc as specified in clause 7.

6.4 Hexadecimal representation

For the hexadecimal format, the octets of the binary format shall be represented by a string of hexadecimal digits, using two hexadecimal digits for each octet of the binary format, the first being the value of the four high-order bits of octet 15, the second being the value of the four low-order bits of octet 15, and so on, with the last being the value of the low-order bits of octet 0 (see 6.5). A HYPHEN-MINUS (45) character (see ISO/IEC 10646) shall be inserted between the hexadecimal representations of each pair of adjacent fields, except between the "VariantAndClockSeqHigh" field and the "ClockSeqLow" field (see the example in clause 8).

6.5 Formal syntax of the hexadecimal representation

6.5.1 The formal definition of the UUID hexadecimal representation syntax is specified using the extended BNF notation defined in Rec. ITU-T X.680 | ISO/IEC 8824-1, clause 5, except that there shall be no white-space between the lexical items.

6.5.2 The "hexdigit" lexical item is used in the BNF specification and is defined as follows:

Name of lexical item – hexdigit

A "hexdigit" shall consist of exactly one of the characters:

```
A B C D E F a b c d e f 0 1 2 3 4 5 6 7 8 9
```

6.5.3 The hexadecimal representation of a UUID shall be the production "UUID":

```
UUID ::=
      TimeLow
      "-" TimeMid
      "-" VersionAndTimeHigh
      "-" VariantAndClockSeqHigh ClockSeqLow
      "-" Node

TimeLow ::=
      HexOctet HexOctet HexOctet HexOctet

TimeMid ::=
      HexOctet HexOctet

VersionAndTimeHigh ::=
      HexOctet HexOctet

VariantAndClockSeqHigh ::=
      HexOctet

ClockSeqLow ::=
      HexOctet

Node ::=
      HexOctet HexOctet HexOctet HexOctet HexOctet HexOctet

HexOctet ::=
      hexdigit hexdigit
```

6.5.4 Software generating the hexadecimal representation of a UUID shall not use upper case letters.

NOTE – It is recommended that the hexadecimal representation used in all human-readable formats be restricted to lower-case letters. Software processing this representation is, however, required to accept both upper and lower case letters as specified in 6.5.2.

7 Use of a UUID as the primary integer value and Unicode label of a Joint UUID arc

7.1 A UUID can be used as the primary integer value of a Joint UUID arc using the single integer value of the UUID specified in 6.3.

7.2 The hexadecimal representation of the UUID defined in 6.4 can also be used as a non-integer Unicode label for the arc.

EXAMPLE – The following is an example of the use of a UUID to form an IRI/URI value:

"oid:/UUID/f81d4fae-7dec-11d0-a765-00a0c91e6bf6"

8 Use of a UUID to form a URN

A URN (see IETF RFC 2141) formed using a UUID shall be the string "**urn:uuid:**" followed by the hexadecimal representation of a UUID defined in 6.4.

EXAMPLE – The following is an example of the string representation of a UUID as a URN:

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

NOTE – An alternative URN format (see [2]) is available, but is not recommended for URNs generated using UUIDs. This alternative format uses the single integer value of the UUID specified in 6.3, and represents the above example as "**urn:oid:2.25.329800735698586629295641978511506172918**".

9 Rules for comparison and ordering of UUIDs

9.1 To compare a pair of UUIDs, the values of the corresponding fields (see 6.1) of each UUID are compared, in order of significance (see 6.1.2). Two UUIDs are equal if and only if all the corresponding fields are equal.

NOTE 1 – This algorithm for comparing two UUIDs is equivalent to the comparison of the values of the single integer representations specified in 6.3.

NOTE 2 – This comparison uses the physical fields specified in 6.1.1 not the values listed in 6.1.3 and specified in clause 12 (Time, Clock Sequence, Variant, Version, and Node).

9.2 A UUID is considered greater than another UUID if it has a larger value for the most significant field in which they differ.

9.3 In a lexicographical ordering of the hexadecimal representation of UUIDs (see 6.4), a larger UUID shall follow a smaller UUID.

10 Validation

Apart from determining if the variant bits are set correctly, and that the Time value used in a time-based UUID is in the future (and therefore not yet assignable), there is no mechanism for determining if a UUID is valid in any real sense, as all possible values can otherwise occur.

11 Variant bits

11.1 The variant bits are the most significant three bits (bits 7, 6, and 5) of octet 7, which is the most significant octet of the "VariantAndClockSeqHigh" field.

11.2 All UUIDs conforming to this Recommendation | International Standard shall have variant bits with bit 7 of octet 7 set to 1 and bit 6 of octet 7 set to 0. Bit 5 of octet 7 is the most significant bit of the Clock Sequence and shall be set in accordance with 12.4.

NOTE – Bit 5 is listed here as a variant bit because its value distinguishes historical formats. Strictly speaking, it is not part of the variant value for this Recommendation | International Standard, which uses only two bits for the variant.

11.3 Table 1 lists, for information, the use of other values of the variant bits.

Table 1 – Use of the variant bits

Bit 7	Bit 6	Bit 5	Description
0	–	–	Reserved to provide NCS backward compatibility
1	0	–	The variant specified in this Recommendation | International Standard
1	1	0	Reserved to provide Microsoft Corporation backward compatibility
1	1	1	Reserved for future use by this Recommendation | International Standard

12 Use of UUID fields and transmission byte order

12.1 General

12.1.1 Table 2 gives the position and summarizes the use of the various UUID fields in the binary representation.

Table 2 – Position and use of UUID fields

Field	Octet # in UUID	Description
"TimeLow"	15-12	The low-order bits of the Time value (32 bits)
"TimeMid"	11-10	The middle bits of the Time value (16 bits)
"VersionAndTimeHigh"	9-8	The Version (4 bits) followed by the high-order bits of the Time value (12 bits)
"VariantAndClockSeqHigh"	7	The Variant bits (2 bits) followed by the high-order bits of the Clock Sequence (6 bits)
"ClockSeqLow"	6	The low-order bits of the Clock Sequence (8 bits)
"Node"	5-0	The Node (see 12.5) (48 bits)

12.1.2 The position of the UUID fields in the binary representation is illustrated in Figure 1.

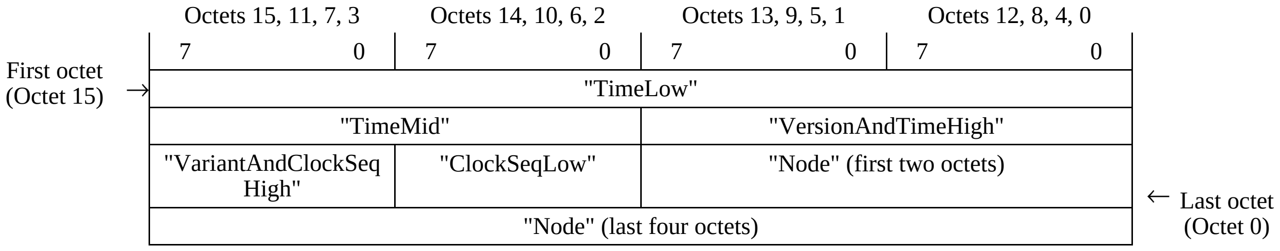

Figure 1 – Position of UUID fields in the binary representation

12.1.3 It is recommended that the binary representation be used for transmission over a communication mechanism, with the sixteen octets of the binary representation transmitted as a contiguous set of sixteen bits with the first octet (octet 15) preceding the last octet (octet 0) in the transmission.

NOTE 1 – The order of the bits within an octet is determined by the communication mechanism specification.

NOTE 2 – The use of sixteen consecutive octets for transmission of a UUID, in the order specified above, is recommended, but protocol specifications may choose alternative means of transferring a UUID, including fragmentation or transmission of only portions of the UUID (such as the parts that contribute to the Time value).

12.2 Version

12.2.1 The three alternative means of generating a UUID (time-based, name-based, and random-number-based) are identified and distinguished by the most significant four bits of the "VersionAndTimeHigh" field (bits 7 to 4 of octet 9 of the UUID). UUIDs generated using these different mechanisms are called "different UUID versions".

NOTE – Describing this as a "different UUID version" is slightly misleading, but the name is used for historical reasons. There is no concept of a traditional "version number" for UUID formats, where new versions may be defined as a revision of this Recommendation | International Standard. Any new UUID format needed in the future would be identified by a different value of the variant bits.

12.2.2 Table 3 lists currently defined "UUID versions", using the first four bits of the "VersionAndTimeHigh" field (bits 7 to 4 of octet 9 of the UUID). It also assigns an integer "Version" value to each combination of bits.

NOTE – A version value of 2 is not used, for compatibility with historical definitions of the UUID. Version values of 0 and 6 to 15 are reserved for future use.

Table 3 – Currently defined UUID versions

Bit 7	Bit 6	Bit 5	Bit 4	Version value	Description
0	0	0	1	1	The time-based version specified in this Recommendation | International Standard (see clause 13)
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 Recommendation | International Standard with MD5 hash (see clause 14)
0	1	0	0	4	The random-number-based version specified in this Recommendation | International Standard (see clause 15)
0	1	0	1	5	The name-based version specified in this Recommendation | International Standard with SHA-1 hash (see clause 14)

12.3 Time

12.3.1 Time shall be a 60-bit value.

NOTE – The name "Time" is appropriate for the time-based version of a UUID (version 1), but is also used for the contents of the corresponding value in the other versions of a UUID (versions 3 and 4).

12.3.2 For the time-based version of a UUID, Time shall be a count of the 100 nanosecond intervals of coordinated universal time (UTC) since the midnight at the start of the 15 October 1582 (the date of the Gregorian reform to the Christian calendar).

NOTE 1 – Before the establishment of the Bureau International de l'Heure (International Time Bureau), every minute contained precisely sixty seconds. Since then, leap seconds have occurred when necessary, increasing (or potentially reducing) the number of seconds per year.

NOTE 2 – Portable systems may have problems determining UTC time, as they are often locked into the local time of their home base. Provided that they continue using the local time of their home base, or change the Clock Sequence value (see 12.4), the UUIDs that they generate will still be unique.

NOTE 3 – For systems that do not have access to broadcast time signals, a system clock recording local time can be used with a time differential added, provided that no UUIDs are generated in the period when a change from daylight saving time occurs, or a change in the Clock Sequence value (see 12.4) is made.

12.3.3 For the name-based version of a UUID, this shall be a 60-bit value constructed from a globally unique name as specified in clause 14.

NOTE – Examples of a globally unique name are OIDs, URNs, and Directory distinguished names (see [1]).

12.3.4 For the random-number-based version of a UUID, this shall be a randomly or pseudo-randomly generated 60-bit value as specified in clause 15.

12.4 Clock sequence

12.4.1 For the time-based version of the UUID, the Clock Sequence is used to help avoid duplicates that could arise when the value of Time is set backwards or if the Node value is changed.

NOTE – The name "Clock Sequence" is appropriate for the time-based version of a UUID, but is also used for the contents of the corresponding value in the name-based and random-number-based versions of a UUID.

12.4.2 If the Time value is set backwards, or might have been set backwards (for example, while the system was powered off), then the UUID generator cannot know whether a UUID has already been generated with Time values larger than the value to which the Time is now set. In such situations, the Clock Sequence value shall be changed.

NOTE – If the previous value of the Clock Sequence is known, it can be just incremented; otherwise it should be set to a cryptographic-quality random or pseudo-random value.

12.4.3 Similarly, if the Node value changes (for example, because a network card has been moved between machines), the Clock Sequence value shall be changed.

12.4.4 The Clock Sequence shall be originally (that is, once in the lifetime of a system producing UUIDs) initialized to a random number that is not derived from the Node value.

NOTE – This is in order to minimize the correlation across systems, providing maximum protection against MAC addresses that may move or switch from system to system rapidly.

12.4.5 For the name-based version of the UUID, the Clock Sequence shall be a 14-bit value constructed from a name as specified in clause 14.

12.4.6 For the random-number-based version of the UUID, the Clock Sequence shall be a randomly or pseudo-randomly generated 14-bit value as specified in clause 15.

12.5 Node

12.5.1 For the time-based version of a UUID, the Node value shall consist of a MAC address (see ISO/IEC 8802-3), usually the host address of some network interface.

12.5.2 For systems with multiple MAC addresses, any available address can be used except a multicast address. Octet 5 of the UUID (the first octet of the "Node") shall be set to the first octet of the MAC address that is transmitted by an ISO/IEC 8802-3-conformant system.

NOTE 1 – This octet contains the global/local bit and the unicast/multicast bit. It is required that the unicast/multicast bit be set to unicast in order to avoid clashes with addresses generated in accordance with 12.5.3.

NOTE 2 – It is possible to obtain a block of MAC addresses from the MAC address registration authority (see [4]).

12.5.3 For systems with no MAC address, a cryptographic-quality random or pseudo-random number may be used (see Annex C). The multicast bit shall be set in such addresses.

NOTE – This is to ensure that the generated addresses never conflict with addresses obtained from network cards as specified in 12.5.2.

12.5.4 For a name-based UUID, the Node value shall be a 48-bit value constructed by canonicalization and hashing from a globally unique name as specified in clause 14.

12.5.5 For a random-number-based UUID, the Node value shall be a randomly or pseudo-randomly generated 48-bit value as specified in clause 15.

13 Setting the fields of a time-based UUID

The fields of a time-based UUID shall be set as follows:

- Determine the values for the UTC-based Time and the Clock Sequence to be used in the UUID, as specified in 12.3 and 12.4.
- For the purposes of this algorithm, consider Time to be a 60-bit unsigned integer and the Clock Sequence to be a 14-bit unsigned integer. Sequentially number the bits in each value, with zero for the least significant bit.
- Set the "TimeLow" field equal to the least significant 32 bits (bits 31 through 0) of Time in the same order of significance.
- Set the "TimeMid" field equal to bits 47 through 32 from the Time in the same order of significance.
- Set the 12 least significant bits (bits 11 through 0) of the "VersionAndTimeHigh" field equal to bits 59 through 48 from Time in the same order of significance.
- Set the four most significant bits (bits 15 through 12) of the "VersionAndTimeHigh" field to the four-bit version number specified in 12.2.
- Set the "ClockSeqLow" field to the eight least significant bits (bits 7 through 0) of the Clock Sequence in the same order of significance.
- Set the six least significant bits (bits 5 through 0) of the "VariantAndClockSeqHigh" field to the six most significant bits (bits 13 through 8) of the Clock Sequence in the same order of significance.
- Set the two most significant bits (bits 7 and 6) of the "VariantAndClockSeqHigh" clock to one and zero, respectively.
- Set the node field to the 48-bit MAC address in the same order of significance as the address.

14 Setting the fields of a name-based UUID

This clause specifies the procedures for the production of a name-based UUID. Subclause 14.1 specifies the general procedures for any hash function (see also ISO/IEC 10118-3). Subclause 14.2 specifies the use of MD5, and 14.3 specifies the use of SHA-1.

NOTE – The use of MD5 is restricted to cases requiring backward compatibility with existing UUIDs, as SHA-1 provides a hashing algorithm with a smaller probability that the same hash value will arise from different hashed material (see C.4).

14.1 The fields of a name-based UUID shall be set as follows:

- Allocate a UUID to use as a "name space identifier" for all UUIDs generated from names in that name space.

  NOTE – Subclause D.9 recommends UUIDs to use for four commonly used name spaces.
- Convert the name to a canonical sequence of octets (as defined by the standards or conventions of its name space).
- Compute the 16-octet hash value of the name space identifier concatenated with the name, using the hash function specified in 14.2 or 14.3. The numbering of the octets in the hash value is from 0 to 15, as specified in IETF RFC 1321 (for MD5) and as specified in FIPS PUB 180-3 (for SHA-1).
- Set octets 3 through 0 of the "TimeLow" field to octets 3 through 0 of the hash value.
- Set octets 1 and 0 of the "TimeMid" field to octets 5 and 4 of the hash value.
- Set octets 1 and 0 of the "VersionAndTimeHigh" field to octets 7 and 6 of the hash value.
- Overwrite the four most significant bits (bits 15 through 12) of the "VersionAndTimeHigh" field with the four-bit version number from Table 3 of 12.2 for the hash function that was used.
- Set the "VariantAndClockSeqHigh" field to octet 8 of the hash value.
- Overwrite the two most significant bits (bits 7 and 6) of the "VariantAndClockSeqHigh" field with 1 and 0, respectively.
- Set the "ClockSeqLow" field to octet 9 of the hash value.
- Set octets 5 through 0 of the "Node" field to octets 15 through 10 of the hash value.

14.2 This subclause specifies a name-based UUID using MD5 as a hash function, but MD5 shall not be used for newly generated UUIDs (see C.4). For an MD5 hash function, the "hash value" referenced in 14.1 is the 16-octet value specified by IETF RFC 1321 as octets zero to 15.

NOTE – This specification of MD5, with the associated version number, is included only for backward compatibility with earlier specifications.

14.3 This subclause specifies a name-based UUID using SHA-1 as a hash function. For a SHA-1 hash function, the "hash value" referenced in 14.1 shall be octets zero to 15 of the 20-octet value obtained from the 160-bit Message Digest value specified by FIPS PUB 180-3. Octets 16 to 19 of the 20-octet value shall be discarded. The 20-octet value shall be obtained from the 160-bit Message Digest value of FIPS PUB 180-3 by placing the most significant bit of the 160-bit value in the most significant bit of the first octet (octet zero) of the 20-octet value, and the least significant bit in the last octet (octet 19) of the 20-octet value.

15 Setting the fields of a random-number-based UUID

15.1 The fields of a random-number-based UUID shall be set as follows:

– Set the two most significant bits (bits 7 and 6) of the "VariantAndClockSeqHigh" field to 1 and 0, respectively.

– Set the four most significant bits (bits 15 through 12) of the "VersionAndTimeHigh" field to the four-bit version number specified in 12.2.

– Set all the other bits of the UUID to randomly (or pseudo-randomly) generated values.

NOTE – Pseudo-random numbers may produce the same value multiple times. The use of cryptographic-quality random numbers is strongly recommended in order to reduce the probability of repeated values.

15.2 Annex C provides guidance on how to generate random numbers in a system.

Annex A

Algorithms for the efficient generation of time-based UUIDs

(This annex does not form an integral part of this Recommendation | International Standard.)

This annex describes an algorithm that can be used to repeatedly generate time-based UUIDs in a computer system.

A.1 Basic algorithm

A.1.1 The following algorithm is simple, correct, but inefficient:

- Obtain a system-wide global lock.
- From a system-wide shared stable store (e.g., a file), read the UUID generator state: the values of the Time, Clock Sequence, and Node used to generate the last UUID.
- Get the current time as a 60-bit count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582 into Time.
- Get the current Node value.
- If the state was unavailable (e.g., non-existent or corrupted), or the saved Node value is different than the current Node value, generate a random Clock Sequence value.
- If the state was available, but the saved Time value is later than the current Time value, increment the Clock Sequence value.
- Save the state (current Time, Clock Sequence, and Node values) back to the stable store.
- Release the global lock.
- Format a UUID from the current Time, Clock Sequence, and Node values according to the steps in clause 13.

A.1.2 If UUIDs do not need to be frequently generated, the above algorithm may be perfectly adequate. For higher performance requirements, however, issues with the basic algorithm include:

- Reading the state from stable storage each time is inefficient;
- The resolution of the system clock may not be 100 nanoseconds;
- Writing the state to stable storage each time is inefficient;
- Sharing the state across process boundaries may be inefficient.

A.1.3 Each of these issues can be addressed in a modular fashion by local improvements in the functions that read and write the state and read the clock. These are addressed in turn in the following subclauses.

A.2 Reading stable storage

A.2.1 The 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).

A.2.2 If an implementation does not have any stable store available, then it can always say that the values were unavailable. This is the least desirable implementation, because it will increase the frequency of creation of new Clock Sequence numbers, which increases the probability of duplicates.

A.2.3 If the Node value can never change (e.g., the net card is inseparable from the system), or if any change also reinitializes the Clock Sequence to a random value, then instead of keeping it in stable store, the current Node value may be returned.

A.3 System clock resolution

A.3.1 The Time value is generated from the system time, whose resolution may be less than the resolution required for Time.

A.3.2 If UUIDs do not need to be frequently generated, Time can simply be the system time multiplied by the number of 100-nanosecond intervals per system time interval.

A.3.3 If a system overruns the generator by requesting too many UUIDs within a single system time interval, the UUID service should either return an error, or stall the UUID generator until the system clock catches up.

A.3.4 A high resolution Time value can be simulated by keeping a count of how many UUIDs have been generated with the same value of the system time, and using it to construct the low-order bits of the Time value. The count will range between zero and the number of 100-nanosecond intervals per system time interval.

NOTE – If the processors overrun the UUID generation frequently, additional MAC addresses can be allocated to the system, which will permit higher speed allocation by making multiple UUIDs potentially available for each Time value.

A.4 Writing stable storage

The state does not always need to be written to stable store every time a UUID is generated. The Time value in the stable store can be periodically set to a value larger than any yet used in a UUID. As long as the generated UUIDs have Time values less than that value, and the Clock Sequence and Node value remain unchanged, only the shared volatile copy of the state needs to be updated. Furthermore, if the Time 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 reinitialization of the Clock Sequence.

A.5 Sharing state across processes

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 Time values each time it is called, and a per-process generator can allocate from that block until it is exhausted.

Annex B

Properties of name-based UUIDs

(This annex does not form an integral part of this Recommendation | International Standard.)

B.1 The name-based UUID is meant for generating a UUID from a name that is drawn from, and unique within, some name space. The concept of name and name space should be broadly construed, and not limited to textual names. The mechanisms or conventions for allocating names from, and ensuring their uniqueness within, their name spaces are beyond the scope of this Specification.

NOTE – In order to avoid recursion problems, name-based UUIDs should not be generated from an OID that ends with a UUID which is name-based.

B.2 The properties of name-based UUIDs generated in accordance with clause 14 and with a suitably chosen namespace will be as follows:

– The UUIDs generated at different times from the same name in the same namespace will be equal;

– The UUIDs generated from two different names in the same namespace will be different with very high probability;

– The UUIDs generated from the same name in two different namespaces will be different with very high probability;

– If two name-based UUIDs are equal, then they were generated from the same name in the same namespace with very high probability.

Annex C

Generation of random numbers in a system

(This annex does not form an integral part of this Recommendation | International Standard.)

C.1 If a system does not have the capability to generate cryptographic-quality random-numbers, then in most systems there are usually a fairly large number of sources of randomness available from which one can be generated. Such sources are system specific, but often include:

– the percent of memory in use;
– the size of main memory in bytes;
– the amount of free main memory in bytes;
– the size of the paging or swap file in bytes;
– free bytes of paging or swap file;
– the total size of user virtual address space in bytes;
– the total available user address space bytes;
– the size of boot disk drive in bytes;
– the free disk space on boot drive in bytes;
– the current time;
– the amount of time since the system booted;
– the individual sizes of files in various system directories;
– the creation, last read, and modification times of files in various system directories;
– the utilization factors of various system resources (heap, etc.);
– current mouse cursor position;
– current caret position;
– current number of running processes, threads;
– handles or IDs of the desktop window and the active window;
– the value of stack pointer of the caller;
– the process and thread identifier of caller;
– various processor architecture specific performance counters (instructions executed, cache misses, Translation Lookaside Buffer or TLB misses).

C.2 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.

C.3 The exact algorithm to generate a Node value using these data is system specific, because both the data available and the functions to obtain them are often very system specific. A generic approach, however, is to accumulate as many sources as possible into a buffer, and use a message digest such as SHA-1, take an arbitrary six octets from the hash value, and set the multicast bit as described above.

C.4 Other hash functions, such as MD5 and hash functions specified in ISO/IEC 10118, can also be used. The only requirement is that the result be suitably random in the sense that the outputs from a set of uniformly distributed inputs are themselves uniformly distributed, and that a single bit change in the input can be expected to cause half of the output bits to change. (The use of MD5, however, is not recommended for new UUIDs because recent research has shown that its output values are not uniformly distributed.)

Annex D

Sample implementation

(This annex does not form an integral part of this Recommendation | International Standard.)

D.1 Files provided

This implementation consists of 6 files: `copyrt.h`, `uuid.h`, `uuid.c`, `sysdep.h`, `sysdep.c` and `utest.c`. The `uuid.*` files are the system independent implementation of the UUID generation algorithms described in clauses 13, 14 and 15, with all the optimizations described in Annex A (except efficient state sharing across processes) included. The code assumes 64-bit integer support, which makes it a lot clearer.

NOTE – The code has been tested on Linux (Red Hat 4.0) with GCC (2.7.2), and Windows NT 4.0 with VC++ 5.0.

D.2 The `copyrt.h` file

All the following source files should be considered to have the following copyright notice included:

```
/*
** Copyright (c) 1990- 1993, 1996 Open Software Foundation, Inc.
** Copyright (c) 1989 by Hewlett-Packard Company, Palo Alto, Ca. &
** Digital Equipment Corporation, Maynard, Mass.
** Copyright (c) 1998 Microsoft.
** To anyone who acknowledges that this file is provided "AS IS"
** without any express or implied warranty: permission to use, copy,
** modify, and distribute this file for any purpose is hereby
** granted without fee, provided that the above copyright notices and
** this notice appears in all source code copies, and that none of
** the names of Open Software Foundation, Inc., Hewlett-Packard
** Company, or Digital Equipment Corporation be used in advertising
** or publicity pertaining to distribution of the software without
** specific, written prior permission. Neither Open Software
** Foundation, Inc., Hewlett-Packard Company, Microsoft, nor Digital
** Equipment Corporation makes any representations about the
** suitability
** of this software for any purpose.
*/
```

D.3 The `uuid.h` file

```
#include "copyrt.h"
#undef uuid_t
typedef struct {
    unsigned32  time_low;
    unsigned16  time_mid;
    unsigned16  time_hi_and_version;
    unsigned8   clock_seq_hi_and_reserved;
    unsigned8   clock_seq_low;
    byte        node[6];
} uuid_t;

/* uuid_create -- generate a UUID */
int uuid_create(uuid_t * uuid);

/* uuid_create_from_name -- create a UUID using a "name"
   from a "name space" */
void uuid_create_from_name(
    uuid_t *uuid,         /* resulting UUID */
    uuid_t nsid,          /* UUID of the namespace */
    void *name,           /* the name from which to generate a UUID */
    int namelen           /* the length of the name */
);

/* uuid_compare --  Compare two UUID's "lexically" and return
        -1   u1 is lexically before u2
         0   u1 is equal to u2
         1   u1 is lexically after u2

   Note that lexical ordering is not temporal ordering!
*/
int uuid_compare(uuid_t *u1, uuid_t *u2);
```

D.4 The `uuid.c` file

```
#include "copyrt.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "sysdep.h"
#ifndef _WINDOWS_
   #include <arpa/inet.h>
#endif
#include "uuid.h"

/* various forward declarations */
static int read_state(unsigned16 *clockseq, uuid_time_t *timestamp,
    uuid_node_t *node);
static void write_state(unsigned16 clockseq, uuid_time_t timestamp,
    uuid_node_t node);
static void format_uuid_v1(uuid_t *uuid, unsigned16 clockseq,
    uuid_time_t timestamp, uuid_node_t node);
static void format_uuid_v3(uuid_t *uuid, unsigned char hash[16]);
static void get_current_time(uuid_time_t *timestamp);
static unsigned16 true_random(void);

/* uuid_create -- generator a UUID */
int uuid_create(uuid_t *uuid)
{
     uuid_time_t timestamp, last_time;
     unsigned16 clockseq;
     uuid_node_t node;
     uuid_node_t last_node;
     int f;

     /* acquire system-wide lock so we're alone */
     LOCK;

     /* get time, node identifier, saved state from non-volatile storage */
     get_current_time(&timestamp);
     get_ieee_node_identifier(&node);
     f = read_state(&clockseq, &last_time, &last_node);

     /* if no NV state, or if clock went backwards, or node identifier
        changed (e.g., new network card) change clockseq */
     if (!f || memcmp(&node, &last_node, sizeof node))
         clockseq = true_random();
     else if (timestamp < last_time)
         clockseq++;

     /* save the state for next time */
     write_state(clockseq, timestamp, node);

     UNLOCK;

     /* stuff fields into the UUID */
     format_uuid_v1(uuid, clockseq, timestamp, node);
     return 1;
}

/* format_uuid_v1 -- make a UUID from the timestamp, clockseq,
                     and node identifier */
void format_uuid_v1(uuid_t* uuid, unsigned16 clock_seq,
                    uuid_time_t timestamp, uuid_node_t node)
{
    /* Construct a version 1 uuid with the information we've gathered
       plus a few constants. */
    uuid->time_low = (unsigned long)(timestamp & 0xFFFFFFFF);
    uuid->time_mid = (unsigned short)((timestamp >> 32) & 0xFFFF);
    uuid->time_hi_and_version =
        (unsigned short)((timestamp >> 48) & 0x0FFF);
    uuid->time_hi_and_version |= (1 << 12);
    uuid->clock_seq_low = clock_seq & 0xFF;
    uuid->clock_seq_hi_and_reserved = (clock_seq & 0x3F00) >> 8;
    uuid->clock_seq_hi_and_reserved |= 0x80;
    memcpy(&uuid->node, &node, sizeof uuid->node);
}

/* data type for UUID generator persistent state */
typedef struct {
    uuid_time_t  ts;       /* saved timestamp */
    uuid_node_t  node;      /* saved node identifier */
    unsigned16   cs;        /* saved Clock Sequence */
} uuid_state;

static uuid_state st;

/* read_state -- read UUID generator state from non-volatile store */
int read_state(unsigned16 *clockseq, uuid_time_t *timestamp,
               uuid_node_t *node)
{
    static int inited = 0;
    FILE *fp;

    /* only need to read state once per boot */
    if (!inited) {
        fp = fopen("state", "rb");
        if (fp == NULL)
            return 0;
        fread(&st, sizeof st, 1, fp);
        fclose(fp);
        inited = 1;
    }
    *clockseq = st.cs;
    *timestamp = st.ts;
    *node = st.node;
    return 1;
}

/* write_state -- save UUID generator state back to non-volatile
   storage */
void write_state(unsigned16 clockseq, uuid_time_t timestamp,
                 uuid_node_t node)
{
    static int inited = 0;
    static uuid_time_t next_save;
    FILE* fp;

    if (!inited) {
        next_save = timestamp;
        inited = 1;
    }

    /* always save state to volatile shared state */
    st.cs = clockseq;
    st.ts = timestamp;
    st.node = node;
    if (timestamp >= next_save) {
        fp = fopen("state", "wb");
        fwrite(&st, sizeof st, 1, fp);
        fclose(fp);
        /* schedule next save for 10 seconds from now */
        next_save = timestamp + (10 * 10 * 1000 * 1000);
    }
}

/* get-current_time -- get time as 60-bit 100ns ticks since UUID epoch.
   Compensate for the fact that real clock resolution is
   less than 100ns. */
void get_current_time(uuid_time_t *timestamp)
{
    static int inited = 0;
    static uuid_time_t time_last;
    static unsigned16 uuids_this_tick;
    uuid_time_t time_now;

    if (!inited) {
        get_system_time(&time_now);
        uuids_this_tick = UUIDS_PER_TICK;
        inited = 1;
    }

    for ( ; ; ) {
        get_system_time(&time_now);

        /* if clock reading changed since last UUID generated, */
        if (time_last != time_now) {
            /* reset count of uuids gen'd with this clock reading */
            uuids_this_tick = 0;
            time_last = time_now;
            break;
        }
        if (uuids_this_tick < UUIDS_PER_TICK) {
            uuids_this_tick++;
            break;
        }
        /* going too fast for our clock; spin */
    }
    /* add the count of uuids to low order bits of the clock reading */
    *timestamp = time_now + uuids_this_tick;
}

/* true_random -- generate a crypto-quality random number.
   **This sample doesn't do that.** */
static unsigned16 true_random(void)
{
    static int inited = 0;
    uuid_time_t time_now;

    if (!inited) {
        get_system_time(&time_now);
        time_now = time_now / UUIDS_PER_TICK;
        srand((unsigned int)(((time_now >> 32) ^ time_now) & 0xffffffff));
        inited = 1;
    }

    return rand();
}

/* uuid_create_from_name -- create a UUID using a "name" from a "name
   space" */
void uuid_create_from_name(uuid_t *uuid, uuid_t nsid, void *name,
                           int namelen)
{
    MD5_CTX c;

    unsigned char hash[16];
    uuid_t net_nsid;

    /* put name space identifier in network byte order so it hashes the
       same no matter what endian machine we're on */
    net_nsid = nsid;
    htonl(net_nsid.time_low);
    htons(net_nsid.time_mid);
    htons(net_nsid.time_hi_and_version);

    MD5Init(&c);
    MD5Update(&c, &net_nsid, sizeof net_nsid);
    MD5Update(&c, name, namelen);
    MD5Final(hash, &c);

    /* the hash is in network byte order at this point */
    format_uuid_v3(uuid, hash);
}

/* format_uuid_v3 -- make a UUID from a (pseudo)random 128-bit number */
void format_uuid_v3(uuid_t *uuid, unsigned char hash[16])
{
    /* convert UUID to local byte order */
    memcpy(uuid, hash, sizeof *uuid);
    ntohl(uuid->time_low);
    ntohs(uuid->time_mid);
    ntohs(uuid->time_hi_and_version);

    /* put in the variant and version bits */
    uuid->time_hi_and_version &= 0x0FFF;
    uuid->time_hi_and_version |= (3 << 12);
    uuid->clock_seq_hi_and_reserved &= 0x3F;
    uuid->clock_seq_hi_and_reserved |= 0x80;
}

/* uuid_compare --  Compare two UUID's "lexically" and return */
#define CHECK(f1, f2) if (f1 != f2) return f1 < f2 ? -1 : 1;
int uuid_compare(uuid_t *u1, uuid_t *u2)
{
    int i;

    CHECK(u1->time_low, u2->time_low);
    CHECK(u1->time_mid, u2->time_mid);
    CHECK(u1->time_hi_and_version, u2->time_hi_and_version);
    CHECK(u1->clock_seq_hi_and_reserved, u2->clock_seq_hi_and_reserved);
    CHECK(u1->clock_seq_low, u2->clock_seq_low)
    for (i = 0; i < 6; i++) {
        if (u1->node[i] < u2->node[i])
            return -1;
        if (u1->node[i] > u2->node[i])
            return 1;
    }
    return 0;
}
#undef CHECK
```

D.5 The `sysdep.h` file

```
#include "copyrt.h"
/* remove the following define if you aren't running Windows 32 */
#define WININC 0

#ifdef WININC
#include <windows.h>
#else
#include <time.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
#endif

#include "global.h"
/* change to point to where MD5 .h's live; IETF RFC 1321 has a sample
   implementation */
#include "md5.h"

/* set the following to the number of 100ns ticks of the actual
   resolution of your system's clock */
#define UUIDS_PER_TICK 1024

/* set the following to a call to get and release a global lock */
#define LOCK
#define UNLOCK

typedef unsigned long   unsigned32;
typedef unsigned short  unsigned16;
typedef unsigned char   unsigned8;
typedef unsigned char   byte;

/* set this to what your compiler uses for 64-bit data type */
#ifdef WININC
#define unsigned64_t unsigned __int64
#define I64(C) C
#else
#define unsigned64_t unsigned long long
#define I64(C) C##LL
#endif

typedef unsigned64_t uuid_time_t;
typedef struct {
    char nodeID[6];
} uuid_node_t;

void get_ieee_node_identifier(uuid_node_t *node);
void get_system_time(uuid_time_t *uuid_time);
void get_random_info(unsigned char seed[16]);
```

D.6 The `sysdep.c` file

```
#include "copyrt.h"
#include <stdio.h>
#include <string.h>
#include "sysdep.h"

/* system dependent call to get MAC node identifier.
   This sample implementation generates a random node identifier. */
void get_ieee_node_identifier(uuid_node_t *node)
{
    static int inited = 0;
    static uuid_node_t saved_node;
    unsigned char seed[16];
    FILE *fp;

    if (!inited) {
        fp = fopen("nodeid", "rb");
        if (fp) {
            fread(&saved_node, sizeof saved_node, 1, fp);
            fclose(fp);
        }
        else {
            get_random_info(seed);
            seed[0] |= 0x80;
            memcpy(&saved_node, seed, sizeof saved_node);
            fp = fopen("nodeid", "wb");
            if (fp) {
                fwrite(&saved_node, sizeof saved_node, 1, fp);
                fclose(fp);
            }
        }
        inited = 1;
    }

    *node = saved_node;
}
/* system dependent call to get the current system time. Returned as
   100ns ticks since UUID epoch, but resolution may be less than 100ns. */
#ifdef _WINDOWS_

void get_system_time(uuid_time_t *uuid_time)
{
    ULARGE_INTEGER time;

    /* Windows NT keeps time in FILETIME format which is 100ns ticks since
       Jan 1, 1601. UUIDs use time in 100ns ticks since Oct 15, 1582.
       The difference is 17 Days in Oct + 30 (Nov) + 31 (Dec)
       + 18 years and 5 leap days. */
    GetSystemTimeAsFileTime((FILETIME *)&time);
    time.QuadPart +=
          (unsigned __int64) (1000*1000*10)       // seconds
        * (unsigned __int64) (60 * 60 * 24)       // days
        * (unsigned __int64) (17+30+31+365*18+5); // # of days
    *uuid_time = time.QuadPart;
}

void get_random_info(unsigned char seed[16])
{
    MD5_CTX c;
    struct {
        MEMORYSTATUS m;
        SYSTEM_INFO s;
        FILETIME t;
        LARGE_INTEGER pc;
        DWORD tc;
        DWORD l;
        char hostname[MAX_COMPUTERNAME_LENGTH + 1];
    } r;

    MD5Init(&c);
    GlobalMemoryStatus(&r.m);
    GetSystemInfo(&r.s);
    GetSystemTimeAsFileTime(&r.t);
    QueryPerformanceCounter(&r.pc);
    r.tc = GetTickCount();
    r.l = MAX_COMPUTERNAME_LENGTH + 1;
    GetComputerName(r.hostname, &r.l);
    MD5Update(&c, &r, sizeof r);
    MD5Final(seed, &c);
}

#else

void get_system_time(uuid_time_t *uuid_time)

{
    struct timeval tp;

    gettimeofday(&tp, (struct timezone *)0);
    /* Offset between UUID formatted times and Unix formatted times.
       UUID UTC base time is October 15, 1582.
       Unix base time is January 1, 1970. */
    *uuid_time = (tp.tv_sec * 10000000) + (tp.tv_usec * 10)
        + I64(0x01B21DD213814000);
}

void get_random_info(unsigned char seed[16])
{
    MD5_CTX c;
    struct {
        struct timeval t;
        char hostname[257];
    } r;

    MD5Init(&c);
    gettimeofday(&r.t, (struct timezone *)0);
    gethostname(r.hostname, 256);
    MD5Update(&c, &r, sizeof r);
    MD5Final(seed, &c);
}

#endif
```

D.7 The `utest.c` file

```
#include "copyrt.h"
#include "sysdep.h"
#include <stdio.h>
#include "uuid.h"

uuid_t NameSpace_DNS = { /* 6ba7b810-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b810,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};

/* puid -- print a UUID */
void puid(uuid_t u)
{
    int i;
    printf("%8.8x-%4.4x-%4.4x-%2.2x%2.2x-", u.time_low, u.time_mid,
    u.time_hi_and_version, u.clock_seq_hi_and_reserved,
    u.clock_seq_low);
    for (i = 0; i < 6; i++)
        printf("%2.2x", u.node[i]);
    printf("\n");
}

/* simple driver for UUID generator */
int main(int argc, char **argv)
{
    uuid_t u;
    int f;

    uuid_create(&u);
    printf("uuid_create(): "); puid(u);

    f = uuid_compare(&u, &u);
    printf("uuid_compare(u,u): %d\n", f);     /* should be 0 */
    f = uuid_compare(&u, &NameSpace_DNS);
    printf("uuid_compare(u, NameSpace_DNS): %d\n", f); /* should be 1 */
    f = uuid_compare(&NameSpace_DNS, &u);
    printf("uuid_compare(NameSpace_DNS, u): %d\n", f); /* should be -1 */
    uuid_create_from_name(&u, NameSpace_DNS, "www.widgets.com", 15);
    printf("uuid_create_from_name(): "); puid(u);
}
```

D.8 Sample output of `utest`

```
uuid_create(): 7d444840-9dc0-11d1-b245-5ffdce74fad2
uuid_compare(u,u): 0
uuid_compare(u, NameSpace_DNS): 1
uuid_compare(NameSpace_DNS, u): -1
uuid_create_from_name(): e902893a-9d22-3c7e-a7b8-d6e313b71d9f
```

D.9 Some name space IDs

This subclause lists the name space IDs for some potentially interesting name spaces, as initialized structures in the C language and in the string representation defined above.

```
/* Name string is a fully-qualified domain name */
uuid_t NameSpace_DNS = { /* 6ba7b810-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b810,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};

/* Name string is a URL */
uuid_t NameSpace_URL = { /* 6ba7b811-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b811,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};

/* Name string is an OID */
uuid_t NameSpace_OID = { /* 6ba7b812-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b812,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};

/* Name string is a Directory distinguished name (in DER or a text output format) */
uuid_t NameSpace_X500 = { /* 6ba7b814-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b814,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};
```

Bibliography

[1] Recommendation ITU-T X.500 (2012) | ISO/IEC 9594-1:2012, *Information technology – Open Systems Interconnection – The Directory: Overview of concepts, models and services.*

[2] IETF RFC 3061 (2001), *A URN Namespace of Object Identifiers*.

[3] IETF RFC 4122 (2005), *A Universally Unique Identifier (UUID) URN Namespace*.

[4] IEEE, Request Form for an Individual Address Block (also known as an Ethernet Address Block) of 4,096 MAC Addresses.
http://standards.ieee.org/develop/regauth/iab/

[5] ISO/IEC 11578:1996, *Information technology – Open Systems Interconnection – Remote Procedure Cal1 (RPC)*.

[6] Open Group CAE: DCE: *Remote Procedure Call*, Specification C309, ISBN 1-85912-041-5, August 1994.

[7] Zahn, L., Dineen, T., Leach, P. (January 1990), *Network Computing Architecture*, ISBN 0-13-611674-4.

Printed in Switzerland
Geneva, 2014