Network Working Group
Request for Comments: 4122
Category: Standards Track

P. Leach
Microsoft
M. Mealling
Refactored Networks, LLC
R. Salz
DataPower Technology, Inc.

July 2005

A Universally Unique IDentifier (UUID) URN Namespace

Status of This Memo

Copyright Notice

Abstract

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

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

Table of Contents

1. Introduction

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

The information here is meant to be a concise guide for those wishing to implement services using UUIDs as URNs. Nothing in this document should be construed to override the DCE standards that defined UUIDs.

There is an ITU-T Recommendation and ISO/IEC Standard [3] that are derived from earlier versions of this document. Both sets of specifications have been aligned, and are fully technically compatible. In addition, a global registration function is being provided by the Telecommunications Standardisation Bureau of ITU-T; for details see <http://www.itu.int/ITU-T/asn1/uuid.html>.

2. Motivation

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

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

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

3. Namespace Registration Template

Namespace ID: UUID
Registration Information:
Registration date: 2003-10-01

Declared registrant of the namespace:
JTC 1/SC6 (ASN.1 Rapporteur Group)

Declaration of syntactic structure:
A UUID is an identifier that is unique across both space and time, with respect to the space of all UUIDs. Since a UUID is a fixed size and contains a time field, it is possible for values to rollover (around A.D. 3400, depending on the specific algorithm used). 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.

The internal representation of a UUID is a specific sequence of bits in memory, as described in Section 4. To accurately represent a UUID as a URN, it is necessary to convert the bit sequence to a string representation.

Each field is treated as an integer and has its value printed as a zero-filled hexadecimal digit string with the most significant digit first. The hexadecimal values "a" through "f" are output as lower case characters and are case insensitive on input.

The formal definition of the UUID string representation is provided by the following ABNF [7]:

```
UUID                   = time-low "-" time-mid "-"
                         time-high-and-version "-"
                         clock-seq-and-reserved
                         clock-seq-low "-" node
time-low               = 4hexOctet
time-mid               = 2hexOctet
time-high-and-version  = 2hexOctet
clock-seq-and-reserved = hexOctet
clock-seq-low          = hexOctet
node                   = 6hexOctet
hexOctet               = hexDigit hexDigit
hexDigit =
      "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" /
      "a" / "b" / "c" / "d" / "e" / "f" /
      "A" / "B" / "C" / "D" / "E" / "F"
```

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

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

Relevant ancillary documentation:
[1][2]

Identifier uniqueness considerations:
This document specifies three algorithms to generate UUIDs: the first leverages the unique values of 802 MAC addresses to guarantee uniqueness, the second uses pseudo-random number generators, and the third uses cryptographic hashing and application-provided text strings. As a result, the UUIDs generated according to the mechanisms here will be unique from all other UUIDs that have been or will be assigned.

Identifier persistence considerations:
UUIDs are inherently very difficult to resolve in a global sense. This, coupled with the fact that UUIDs are temporally unique within their spatial context, ensures that UUIDs will remain as persistent as possible.

Process of identifier assignment:
Generating a UUID does not require that a registration authority be contacted. One algorithm requires a unique value over space for each generator. This value is typically an IEEE 802 MAC address, usually already available on network-connected hosts. The address can be assigned from an address block obtained from the IEEE registration authority. If no such address is available,

or privacy concerns make its use undesirable, Section 4.5 specifies two alternatives. Another approach is to use version 3 or version 4 UUIDs as defined below.

Process for identifier resolution:
Since UUIDs are not globally resolvable, this is not applicable.

Rules for Lexical Equivalence:
Consider each field of the UUID to be an unsigned integer as shown in the table in section Section 4.1.2. Then, to compare a pair of UUIDs, arithmetically compare the corresponding fields from each UUID in order of significance and according to their data type. Two UUIDs are equal if and only if all the corresponding fields are equal.

As an implementation note, equality comparison can be performed on many systems by doing the appropriate byte-order canonicalization, and then treating the two UUIDs as 128-bit unsigned integers.

UUIDs, as defined in this document, can also be ordered lexicographically. For a pair of UUIDs, the first one follows the second if the most significant field in which the UUIDs differ is greater for the first UUID. The second precedes the first if the most significant field in which the UUIDs differ is greater for the second UUID.

Conformance with URN Syntax:
The string representation of a UUID is fully compatible with the URN syntax. When converting from a bit-oriented, in-memory representation of a UUID into a URN, care must be taken to strictly adhere to the byte order issues mentioned in the string representation section.

Validation mechanism:
Apart from determining whether the timestamp portion of the UUID is in the future and therefore not yet assignable, there is no mechanism for determining whether a UUID is 'valid'.

Scope:
UUIDs are global in scope.

4. Specification

4.1. Format

The UUID format is 16 octets; some bits of the eight octet variant field specified below determine finer structure.

4.1.1. Variant

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

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

Msb0	Msb1	Msb2	Description
0	x	x	Reserved, NCS backward compatibility.
1	0	x	The variant specified in this document.
1	1	0	Reserved, Microsoft Corporation backward compatibility
1	1	1	Reserved for future definition.

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

4.1.2. Layout and Byte Order

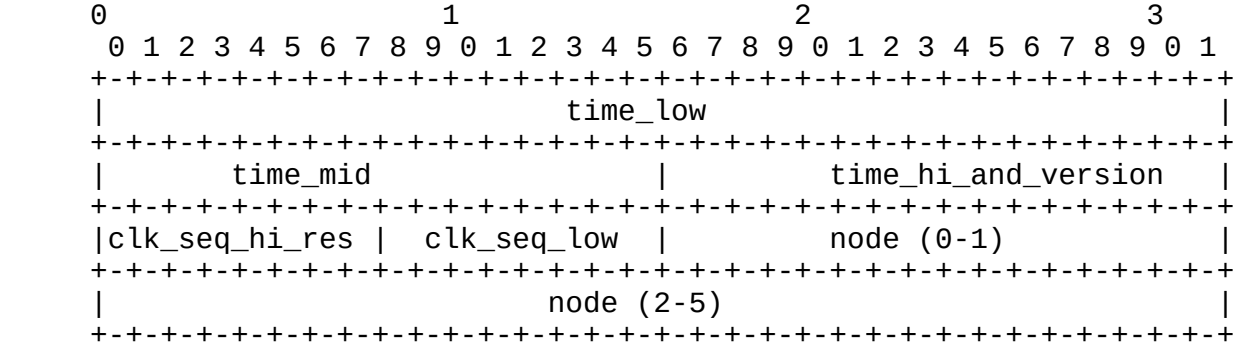
To minimize confusion about bit assignments within octets, the UUID record definition is defined only in terms of fields that are integral numbers of octets. The fields are presented with the most significant one first.

Field	Data Type	Octet #	Note
time_low	unsigned 32 bit integer	0-3	The low field of the timestamp
time_mid	unsigned 16 bit integer	4-5	The middle field of the timestamp
time_hi_and_version	unsigned 16 bit integer	6-7	The high field of the timestamp multiplexed with the version number

clock_seq_hi_and_reserved	unsigned 8 bit integer	8	The high field of the clock sequence multiplexed with the variant
clock_seq_low	unsigned 8 bit integer	9	The low field of the clock sequence
node	unsigned 48 bit integer	10-15	The spatially unique node identifier

In the absence of explicit application or presentation protocol specification to the contrary, a UUID is encoded as a 128-bit object, as follows:

The fields are encoded as 16 octets, with the sizes and order of the fields defined above, and with each field encoded with the Most Significant Byte first (known as network byte order). Note that the field names, particularly for multiplexed fields, follow historical practice.



4.1.3. Version

The version number is in the most significant 4 bits of the time stamp (bits 4 through 7 of the time_hi_and_version field).

The following table lists the currently-defined versions for this UUID variant.

Msb0	Msb1	Msb2	Msb3	Version	Description
0	0	0	1	1	The time-based version specified in this document.
0	0	1	0	2	DCE Security version, with embedded POSIX UIDs.

0	0	1	1	3	The name-based version specified in this document that uses MD5 hashing.
0	1	0	0	4	The randomly or pseudo-randomly generated version specified in this document.
0	1	0	1	5	The name-based version specified in this document that uses SHA-1 hashing.

The version is more accurately a sub-type; again, we retain the term for compatibility.

4.1.4. Timestamp

The timestamp is a 60-bit value. For UUID version 1, this is represented by Coordinated Universal Time (UTC) as a count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582 (the date of Gregorian reform to the Christian calendar).

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

For UUID version 3 or 5, the timestamp is a 60-bit value constructed from a name as described in Section 4.3.

For UUID version 4, the timestamp is a randomly or pseudo-randomly generated 60-bit value, as described in Section 4.4.

4.1.5. Clock Sequence

For UUID version 1, the clock sequence is used to help avoid duplicates that could arise when the clock is set backwards in time or if the node ID changes.

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

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

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

For UUID version 3 or 5, the clock sequence is a 14-bit value constructed from a name as described in Section 4.3.

For UUID version 4, clock sequence is a randomly or pseudo-randomly generated 14-bit value as described in Section 4.4.

4.1.6. Node

For UUID version 1, the node field consists of an IEEE 802 MAC address, usually the host address. For systems with multiple IEEE 802 addresses, any available one can be used. The lowest addressed octet (octet number 10) contains the global/local bit and the unicast/multicast bit, and is the first octet of the address transmitted on an 802.3 LAN.

For systems with no IEEE address, a randomly or pseudo-randomly generated value may be used; see Section 4.5. The multicast bit must be set in such addresses, in order that they will never conflict with addresses obtained from network cards.

For UUID version 3 or 5, the node field is a 48-bit value constructed from a name as described in Section 4.3.

For UUID version 4, the node field is a randomly or pseudo-randomly generated 48-bit value as described in Section 4.4.

4.1.7. Nil UUID

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

4.2. Algorithms for Creating a Time-Based UUID

Various aspects of the algorithm for creating a version 1 UUID are discussed in the following sections.

4.2.1. Basic Algorithm

The following algorithm is simple, correct, and 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 timestamp, clock sequence, and node ID 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.
- Get the current node ID.
- If the state was unavailable (e.g., non-existent or corrupted), or the saved node ID is different than the current node ID, generate a random clock sequence value.
- If the state was available, but the saved timestamp is later than the current timestamp, increment the clock sequence value.
- Save the state (current timestamp, clock sequence, and node ID) back to the stable store.
- Release the global lock.
- Format a UUID from the current timestamp, clock sequence, and node ID values according to the steps in Section 4.2.2.

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.

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. We address each of them in turn in the following sections.

4.2.1.1. Reading Stable Storage

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).

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.

If the node ID 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 ID may be returned.

4.2.1.2. System Clock Resolution

The timestamp is generated from the system time, whose resolution may be less than the resolution of the UUID timestamp.

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

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

A high resolution timestamp can be simulated by keeping a count of the number of UUIDs that have been generated with the same value of the system time, and using it to construct the low order bits of the timestamp. 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 node identifiers can be allocated to the system, which will permit higher speed allocation by making multiple UUIDs potentially available for each time stamp value.

4.2.1.3. Writing Stable Storage

The state does not always need to be written to stable store every time a UUID is generated. The timestamp in the stable store can be periodically set to a value larger than any yet used in a UUID. As long as the generated UUIDs have timestamps less than that value, and

the clock sequence and node ID remain unchanged, only the shared volatile copy of the state needs to be updated. Furthermore, if the timestamp value in stable store is in the future by less than the typical time it takes the system to reboot, a crash will not cause a reinitialization of the clock sequence.

4.2.1.4. 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 stamps each time it is called; a per-process generator can allocate from that block until it is exhausted.

4.2.2. Generation Details

Version 1 UUIDs are generated according to the following algorithm:

- Determine the values for the UTC-based timestamp and clock sequence to be used in the UUID, as described in Section 4.2.1.
- For the purposes of this algorithm, consider the timestamp to be a 60-bit unsigned integer and the clock sequence to be a 14-bit unsigned integer. Sequentially number the bits in a field, starting with zero for the least significant bit.
- Set the time_low field equal to the least significant 32 bits (bits zero through 31) of the timestamp in the same order of significance.
- Set the time_mid field equal to bits 32 through 47 from the timestamp in the same order of significance.
- Set the 12 least significant bits (bits zero through 11) of the time_hi_and_version field equal to bits 48 through 59 from the timestamp in the same order of significance.
- Set the four most significant bits (bits 12 through 15) of the time_hi_and_version field to the 4-bit version number corresponding to the UUID version being created, as shown in the table above.
- Set the clock_seq_low field to the eight least significant bits (bits zero through 7) of the clock sequence in the same order of significance.

- Set the 6 least significant bits (bits zero through 5) of the clock_seq_hi_and_reserved field to the 6 most significant bits (bits 8 through 13) of the clock sequence in the same order of significance.

- Set the two most significant bits (bits 6 and 7) of the clock_seq_hi_and_reserved to zero and one, respectively.

- Set the node field to the 48-bit IEEE address in the same order of significance as the address.

4.3. Algorithm for Creating a Name-Based UUID

The version 3 or 5 UUID is meant for generating UUIDs from "names" that are 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. For example, some name spaces are the domain name system, URLs, ISO Object IDs (OIDs), X.500 Distinguished Names (DNs), and reserved words in a programming language. The mechanisms or conventions used for allocating names and ensuring their uniqueness within their name spaces are beyond the scope of this specification.

The requirements for these types of UUIDs are as follows:

- The UUIDs generated at different times from the same name in the same namespace MUST be equal.

- The UUIDs generated from two different names in the same namespace should be different (with very high probability).

- The UUIDs generated from the same name in two different namespaces should be different with (very high probability).

- If two UUIDs that were generated from names are equal, then they were generated from the same name in the same namespace (with very high probability).

The algorithm for generating a UUID from a name and a name space are as follows:

- Allocate a UUID to use as a "name space ID" for all UUIDs generated from names in that name space; see Appendix C for some pre-defined values.

- Choose either MD5 [4] or SHA-1 [8] as the hash algorithm; If backward compatibility is not an issue, SHA-1 is preferred.

- Convert the name to a canonical sequence of octets (as defined by the standards or conventions of its name space); put the name space ID in network byte order.
- Compute the hash of the name space ID concatenated with the name.
- Set octets zero through 3 of the time_low field to octets zero through 3 of the hash.
- Set octets zero and one of the time_mid field to octets 4 and 5 of the hash.
- Set octets zero and one of the time_hi_and_version field to octets 6 and 7 of the hash.
- Set the four most significant bits (bits 12 through 15) of the time_hi_and_version field to the appropriate 4-bit version number from Section 4.1.3.
- Set the clock_seq_hi_and_reserved field to octet 8 of the hash.
- Set the two most significant bits (bits 6 and 7) of the clock_seq_hi_and_reserved to zero and one, respectively.
- Set the clock_seq_low field to octet 9 of the hash.
- Set octets zero through five of the node field to octets 10 through 15 of the hash.
- Convert the resulting UUID to local byte order.

4.4. Algorithms for Creating a UUID from Truly Random or Pseudo-Random Numbers

The version 4 UUID is meant for generating UUIDs from truly-random or pseudo-random numbers.

The algorithm is as follows:

- Set the two most significant bits (bits 6 and 7) of the clock_seq_hi_and_reserved to zero and one, respectively.
- Set the four most significant bits (bits 12 through 15) of the time_hi_and_version field to the 4-bit version number from Section 4.1.3.
- Set all the other bits to randomly (or pseudo-randomly) chosen values.

See Section 4.5 for a discussion on random numbers.

4.5. Node IDs that Do Not Identify the Host

This section describes how to generate a version 1 UUID if an IEEE 802 address is not available, or its use is not desired.

One approach is to contact the IEEE and get a separate block of addresses. At the time of writing, the application could be found at <http://standards.ieee.org/regauth/oui/pilot-ind.html>, and the cost was US$550.

A better solution is to obtain a 47-bit cryptographic quality random number and use it as the low 47 bits of the node ID, with the least significant bit of the first octet of the node ID set to one. This bit is the unicast/multicast bit, which will never be set in IEEE 802 addresses obtained from network cards. Hence, there can never be a conflict between UUIDs generated by machines with and without network cards. (Recall that the IEEE 802 spec talks about transmission order, which is the opposite of the in-memory representation that is discussed in this document.)

For compatibility with earlier specifications, note that this document uses the unicast/multicast bit, instead of the arguably more correct local/global bit.

Advice on generating cryptographic-quality random numbers can be found in RFC1750 [5].

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

The exact algorithm to generate a node ID using these data is system specific, because both the data available and the functions to obtain them are often very system specific. A generic approach, however, is to accumulate as many sources as possible into a buffer, use a message digest such as MD5 [4] or SHA-1 [8], take an arbitrary 6 bytes from the hash value, and set the multicast bit as described above.

5. Community Considerations

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

This specification attempts to standardize that practice as openly as possible and in a way that attempts to benefit the entire Internet.

6. Security Considerations

Do not assume that UUIDs are hard to guess; they should not be used as security capabilities (identifiers whose mere possession grants access), for example. A predictable random number source will exacerbate the situation.

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

Distributed applications generating UUIDs at a variety of hosts must be willing to rely on the random number source at all hosts. If this is not feasible, the namespace variant should be used.

7. Acknowledgments

This document draws heavily on the OSF DCE specification for UUIDs. Ted Ts'o provided helpful comments, especially on the byte ordering section which we mostly plagiarized from a proposed wording he supplied (all errors in that section are our responsibility, however).

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

8. Normative References

[1] Zahn, L., Dineen, T., and P. Leach, "Network Computing Architecture", ISBN 0-13-611674-4, January 1990.

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

[3] ISO/IEC 9834-8:2004 Information Technology, "Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components" ITU-T Rec. X.667, 2004.

[4] Rivest, R., "The MD5 Message-Digest Algorithm ", RFC 1321, April 1992.

[5] Eastlake, D., 3rd, Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[6] Moats, R., "URN Syntax", RFC 2141, May 1997.

[7] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[8] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-1, April 1995, <http://www.itl.nist.gov/fipspubs/fip180-1.htm>.

Appendix A. Appendix A - Sample Implementation

This implementation consists of 5 files: 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 above, with all the optimizations described above except efficient state sharing across processes included. 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. The code assumes 64-bit integer support, which makes it much clearer.

All the following source files should have the following copyright notice included:

copyrt.h

```
/*
** 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, Microsoft, 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.
*/
```

uuid.h

```
#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_md5_from_name -- create a version 3 (MD5) UUID using a
   "name" from a "name space" */
void uuid_create_md5_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_create_sha1_from_name -- create a version 5 (SHA-1) UUID
   using a "name" from a "name space" */
void uuid_create_sha1_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);
```

uuid.c

```
#include "copyrt.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "sysdep.h"
#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_v3or5(uuid_t *uuid, unsigned char hash[16],
    int v);
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 ID, 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 ID
        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 ID */
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 ID */
    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_md5_from_name -- create a version 3 (MD5) UUID using a
   "name" from a "name space" */
void uuid_create_md5_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 ID in network byte order so it hashes the same
       no matter what endian machine we're on */
    net_nsid = nsid;
    net_nsid.time_low = htonl(net_nsid.time_low);
    net_nsid.time_mid = htons(net_nsid.time_mid);
    net_nsid.time_hi_and_version = 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_v3or5(uuid, hash, 3);
}
```

```
void uuid_create_sha1_from_name(uuid_t *uuid, uuid_t nsid, void *name,
                                int namelen)
{
    SHA_CTX c;
    unsigned char hash[20];
    uuid_t net_nsid;

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

    SHA1_Init(&c);
    SHA1_Update(&c, &net_nsid, sizeof net_nsid);
    SHA1_Update(&c, name, namelen);
    SHA1_Final(hash, &c);

    /* the hash is in network byte order at this point */
    format_uuid_v3or5(uuid, hash, 5);
}

/* format_uuid_v3or5 -- make a UUID from a (pseudo)random 128-bit
   number */
void format_uuid_v3or5(uuid_t *uuid, unsigned char hash[16], int v)
{
    /* convert UUID to local byte order */
    memcpy(uuid, hash, sizeof *uuid);
    uuid->time_low = ntohl(uuid->time_low);
    uuid->time_mid = ntohs(uuid->time_mid);
    uuid->time_hi_and_version = ntohs(uuid->time_hi_and_version);

    /* put in the variant and version bits */
    uuid->time_hi_and_version &= 0x0FFF;
    uuid->time_hi_and_version |= (v << 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
```

sysdep.h

```
#include "copyrt.h"
/* remove the following define if you aren't running WIN32 */
#define WININC 0

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

#include "global.h"
/* change to point to where MD5 .h's live; RFC 1321 has 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 calls 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(char seed[16]);
```

sysdep.c

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

/* system dependent call to get IEEE node ID.
   This sample implementation generates a random node ID. */
void get_ieee_node_identifier(uuid_node_t *node)
{
    static inited = 0;
    static uuid_node_t saved_node;
    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] |= 0x01;
            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;

    /* 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;
}

/* Sample code, not for use in production; see RFC 1750 */
void get_random_info(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 = ((unsigned64)tp.tv_sec * 10000000)
        + ((unsigned64)tp.tv_usec * 10)
        + I64(0x01B21DD213814000);
}

/* Sample code, not for use in production; see RFC 1750 */
void get_random_info(char seed[16])
{
    MD5_CTX c;
    struct {
        struct sysinfo s;
        struct timeval t;
        char hostname[257];
    } r;

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

#endif
```

utest.c

```
#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 */
void 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); /* s.b. 1 */
    f = uuid_compare(&NameSpace_DNS, &u);
    printf("uuid_compare(NameSpace_DNS, u): %d\n", f); /* s.b. -1 */
    uuid_create_md5_from_name(&u, NameSpace_DNS, "www.widgets.com", 15);
    printf("uuid_create_md5_from_name(): "); puid(u);
}
```

Appendix B. Appendix B - 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_md5_from_name(): e902893a-9d22-3c7e-a7b8-d6e313b71d9f
```

Appendix C. Appendix C - Some Name Space IDs

This appendix lists the name space IDs for some potentially interesting name spaces, as initialized C structures 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 ISO OID */
uuid_t NameSpace_OID = { /* 6ba7b812-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b812,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};

/* Name string is an X.500 DN (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
};
```

Authors' Addresses

Paul J. Leach
Microsoft
1 Microsoft Way
Redmond, WA 98052
US

Phone: +1 425-882-8080
EMail: paulle@microsoft.com

Michael Mealling
Refactored Networks, LLC
1635 Old Hwy 41
Suite 112, Box 138
Kennesaw, GA 30152
US

Phone: +1-678-581-9656
EMail: michael@refactored-networks.com
URI: http://www.refactored-networks.com

Rich Salz
DataPower Technology, Inc.
1 Alewife Center
Cambridge, MA 02142
US

Phone: +1 617-864-0455
EMail: rsalz@datapower.com
URI: http://www.datapower.com

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.