
Network Working Group
INTERNET-DRAFT
<draft-leach-uuids-guids-01.txt>
Category: Standards Track
Expires August 4, 1998
February 4, 1998

Paul J. Leach, Microsoft
Rich Salz, Certco

UUIDs and GUIDs

STATUS OF THIS MEMO

ABSTRACT

This specification defines the format of UUIDs (Universally Unique IDentifier), also known as GUIDs (Globally Unique IDentifier). A UUID is 128 bits long, and if generated according to the one of the mechanisms in this document, is either guaranteed to be different from all other UUIDs/GUIDs generated until 3400 A.D. or extremely likely to be different (depending on the mechanism chosen). UUIDs were originally used in the Network Computing System (NCS) [1] and later in the Open Software Foundation's (OSF) Distributed Computing Environment [2].

This specification is derived from the latter specification with the kind permission of the OSF.

Table of Contents

1. Introduction

This specification defines the format of UUIDs (Universally Unique IDentifiers), also known as GUIDs (Globally Unique IDentifiers). A UUID is 128 bits long, and if generated according to the one of the mechanisms in this document, is either guaranteed to be different from all other UUIDs/GUIDs generated until 3400 A.D. or extremely likely to be different (depending on the mechanism chosen).

2. Motivation

One of the main reasons for using UUIDs is that no centralized authority is required to administer them (beyond the one that allocates IEEE 802.1 node identifiers). As a result, generation on demand can be completely automated, and they can be used for a wide variety of purposes. The UUID generation algorithm described here supports very high allocation rates: 10 million per second per machine if you need it, so that they could even be used as transaction IDs.

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

3. Specification

A UUID is an identifier that is unique across both space and time, with respect to the space of all UUIDs. To be precise, the UUID consists of a finite bit space. Thus the time value used for constructing a UUID is limited and will roll over in the future (approximately at A.D. 3400, based on the specified algorithm). 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 generation of UUIDs does not require that a registration authority be contacted for each identifier. Instead, it requires a unique value over space for each UUID generator. This spatially unique value is specified as an IEEE 802 address, which is usually already available to network-connected systems. This 48-bit address can be assigned based on an address block obtained through the IEEE registration authority. This section of the UUID specification assumes the availability of an IEEE 802 address to a system desiring to generate a UUID, but if one is not available section 4 specifies a way to generate a probabilistically unique one that can not conflict with any properly assigned IEEE 802 address.

3.1 Format

In its most general form, all that can be said of the UUID format is that a UUID is 16 octets, and that some bits of octet 8 of the UUID called the variant field (specified in the next section) determine finer structure.

3.1.1 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. The variant field consists of a variable number of the msbs of octet 8 of the UUID.

The following table lists the contents of the variant field.

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

Other UUID variants may not interoperate with the UUID variant specified in this document, where interoperability is defined as the applicability of operations such as string conversion and lexical ordering across different systems. However, UUIDs allocated according to the stricture of different variants, though they may define different interpretations of the bits outside the variant field, will not result in duplicate UUID allocation, because of the differing values of the variant field itself.

The remaining fields described below (version, timestamp, etc.) are defined only for the UUID variant noted above.

3.1.2 UUID layout

The following table gives the format of a UUID for the variant specified herein. The UUID consists of a record of 16 octets. 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 in order of significance for comparison purposes, with "time_low" the most significant, and "node" the least significant.

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.

3.1.3 Version

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

The following table lists currently defined versions of the UUID.

Msb0	Msb1	Msb2	Msb3	Version	Description
0	0	0	1	1	The time-based version specified in this document.
0	0	1	0	2	Reserved for DCE Security version, with embedded POSIX UIDs.
0	0	1	1	3	The name-based version specified in this

					document
0	1	0	0	4	The randomly or pseudo-randomly generated version specified in this document

3.1.4 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 local time, they MAY use local time instead of UTC, as long as they do so consistently throughout the system. This is NOT RECOMMENDED, however, and it should be noted that all that is needed to generate UTC, given local time, is a time zone offset.

For UUID version 3, it is a 60 bit value constructed from a name.

For UUID version 4, it is a randomly or pseudo-randomly generated 60 bit value.

3.1.5 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 even 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 be just 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, it is a 14 bit value constructed from a name.

For UUID version 4, it is a randomly or pseudo-randomly generated 14 bit value.

3.1.6 Node

For UUID version 1, the node field consists of the IEEE address, usually the host address. For systems with multiple IEEE 802 addresses, any available address 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). 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, the node field is a 48 bit value constructed from a name.

For UUID version 4, the node field is a randomly or pseudo-randomly generated 48 bit value.

3.1.7 Nil UUID

The nil UUID is special form of UUID that is specified to have all 128 bits set to 0 (zero).

3.2 Algorithms for creating a time-based UUID

Various aspects of the algorithm for creating a version 1 UUID are discussed in the following sections. UUID generation requires a guarantee of uniqueness within the node ID for a given variant and version. Interoperability is provided by complying with the specified data structure.

3.2.1 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 time stamp, 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 (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 time stamp is later than the current time stamp, increment the clock sequence value

. Format a UUID from the current time stamp, clock sequence, and node ID values according to the structure in section 3.1 (see section 3.2.6 for more details)

. Save the state (current time stamp, clock sequence, and node ID) back to the stable store

. Release the system wide global lock

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.

3.2.2 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.

3.2.3 System clock resolution

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

If UUIDs do not need to be frequently generated, the time stamp 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 time stamp 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 construction the low-order bits of the time stamp. 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.

3.2.4 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 time stamps 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 time stamp 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.

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

3.2.6 UUID Generation details

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 above.

- 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 from 0 (zero) for the least significant bit.

- Set the time_low field equal to the least significant 32-bits (bits numbered 0 to 31 inclusive) of the time stamp in the same order of significance.

- Set the time_mid field equal to the bits numbered 32 to 47 inclusive of the time stamp in the same order of significance.

- Set the 12 least significant bits (bits numbered 0 to 11 inclusive) of the time_hi_and_version field equal to the bits numbered 48 to 59 inclusive of the time stamp in the same order of significance.

- Set the 4 most significant bits (bits numbered 12 to 15 inclusive) 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 in section 3.1.3.

- Set the clock_seq_low field to the 8 least significant bits (bits numbered 0 to 7 inclusive) of the clock sequence in the same order of significance.

- Set the 6 least significant bits (bits numbered 0 to 5 inclusive) of the clock_seq_hi_and_reserved field to the 6 most significant bits (bits numbered 8 to 13 inclusive) of the clock sequence in the same order of significance.

- Set the 2 most significant bits (bits numbered 6 and 7) of the clock_seq_hi_and_reserved to 0 and 1, respectively.

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

3.3 Algorithm for creating a name-based UUID

The version 3 UUID is meant for generating UUIDs from "names" that are drawn from, and unique within, some "name space". Some examples of names (and, implicitly, name spaces) might be DNS names, URLs, ISO Object IDs (OIDs), reserved words in a programming language, or X.500 Distinguished Names (DNs); thus, 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.

The requirements for such 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 the 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
- 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 MD5 [3] hash of the name space ID concatenated with the name
- Set octets 0-3 of time_low field to octets 0-3 of the MD5 hash
- Set octets 0-1 of time_mid field to octets 4-5 of the MD5 hash
- Set octets 0-1 of time_hi_and_version field to octets 6-7 of the MD5 hash
- Set the clock_seq_hi_and_reserved field to octet 8 of the MD5 hash
- Set the clock_seq_low field to octet 9 of the MD5 hash
- Set octets 0-5 of the node field to octets 10-15 of the MD5 hash
- Set the 2 most significant bits (bits numbered 6 and 7) of the clock_seq_hi_and_reserved to 0 and 1, respectively.
- Set the 4 most significant bits (bits numbered 12 to 15 inclusive) 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.
- Convert the resulting UUID to local byte order.

3.4 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 2 most significant bits (bits numbered 6 and 7) of the clock_seq_hi_and_reserved to 0 and 1, respectively.
- Set the 4 most significant bits (bits numbered 12 to 15 inclusive) 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 all the other bits to randomly (or pseudo-randomly) chosen values.

Here are several possible ways to generate the random values:

- Use a physical source of randomness: for example, a white noise generator, radioactive decay, or a lava lamp.

- Use a cryptographic strength random number generator.

3.5 String Representation of UUIDs

For use in human readable text, a UUID string representation is specified as a sequence of fields, some of which are separated by single dashes.

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 to f inclusive are output as lower case characters, and are case insensitive on input. The sequence is the same as the UUID constructed type.

The formal definition of the UUID string representation is provided by the following extended BNF:

```
UUID                   = <time_low> "-" <time_mid> "-"
                         <time_high_and_version> "-"
                         <clock_seq_and_reserved>
                         <clock_seq_low> "-" <node>
time_low               = 4*<hexOctet>
time_mid               = 2*<hexOctet>
time_high_and_version  = 2*<hexOctet>
clock_seq_and_reserved = <hexOctet>
clock_seq_low          = <hexOctet>
node                   = 6*<hexOctet
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:

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

3.6 Comparing UUIDs for equality

Consider each field of the UUID to be an unsigned integer as shown in the table in section 3.1. 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.

Note: as a practical matter, on many systems comparison of two UUIDs for equality can be performed simply by comparing the 128 bits of their in-memory representation considered as a 128 bit unsigned integer. Here, it is presumed that by the time the in-memory representation is obtained the appropriate byte-order canonicalizations have been carried out.

3.7 Comparing UUIDs for relative order

Two UUIDs allocated according to the same variant can also be ordered lexicographically. For the UUID variant herein defined, the first of two UUIDs follows the second if the most significant field in which the UUIDs differ is greater for the first UUID. The first of a pair of UUIDs precedes the second if the most significant field in which the UUIDs differ is greater for the second UUID.

3.8 Byte order of UUIDs

UUIDs may be transmitted in many different forms, some of which may be dependent on the presentation or application protocol where the UUID may be used. In such cases, the order, sizes and byte orders of the UUIDs fields on the wire will depend on the relevant presentation or application protocol. However, it is strongly RECOMMENDED that the order of the fields conform with ordering set out in section 3.1 above. Furthermore, the payload size of each field in the application or presentation protocol MUST be large enough that no information lost in the process of encoding them for transmission.

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 in section 3.1, and with each field encoded with the Most Significant Byte first (also known as network byte order).

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          time_low                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       time_mid                |         time_hi_and_version   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|clk_seq_hi_res |  clk_seq_low  |         node (0-1)            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         node (2-5)                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4. Node IDs when no IEEE 802 network card is available

If a system wants to generate UUIDs but has no IEE 802 compliant network card or other source of IEEE 802 addresses, then this section describes how to generate one.

The ideal 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 most significant bit of the first octet of the node ID set to 1. 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.

If a system does not have a primitive 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 ID of caller
- various processor architecture specific performance counters (instructions executed, cache misses, TLB misses)

(Note that it precisely the above kinds of sources of randomness that are used to seed cryptographic quality random number generators on systems without special hardware for their construction.)

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

The exact algorithm to generate a node ID using these data is system specific, because both the data available and the functions to obtain them are often very system specific. However, assuming that one can concatenate all the values from the randomness sources into a buffer, and that a cryptographic hash function such as MD5 [3] is available, then any 6 bytes of the MD5 hash of the buffer, with the multicast bit (the high bit of the first byte) set will be an appropriately random node ID.

Other hash functions, such as SHA-1 [4], can also be used. The only requirement is that the result be suitably random _ in the sense that the outputs from a set 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.

5. Obtaining IEEE 802 addresses

At the time of writing, the following URL

    http://standards.ieee.org/db/oui/forms/

contains information on how to obtain an IEEE 802 address block. At the time of writing, the cost is $1250 US.

6. Security Considerations

It should not be assumed that UUIDs are hard to guess; they should not be used as capabilities.

7. Acknowledgements

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).

8. References

[1] Lisa Zahn, et. al., Network Computing Architecture, Prentice Hall, Englewood Cliffs, NJ, 1990

[2] DCE: Remote Procedure Call, Open Group CAE Specification C309 ISBN 1-85912-041-5 28cm. 674p. pbk. 1,655g. 8/94

[3] R. Rivest, RFC 1321, "The MD5 Message-Digest Algorithm", 04/16/1992.

[4] NIST FIPS PUB 180-1, "Secure Hash Standard," National Institute of Standards and Technology, U.S. Department of Commerce, DRAFT, May 31, 1994.

9. Authors' addresses

Paul J. Leach
Microsoft
1 Microsoft Way
Redmond, WA, 98052, U.S.A.
paulle@microsoft.com
Tel. 425 882 8080
Fax. 425 936 7329

Rich Salz
100 Cambridge Park Drive
Cambridge MA 02140
salzr@certco.com
Tel. 617 499 4075
Fax. 617 576 0019

10. Notice

11. Full Copyright Statement

Appendix A _ UUID 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 a lot clearer.

All the following source files should be considered to 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, 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 _uuid_t {
    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 to serve as context, so identical
                           names from different name spaces generate
                           different UUIDs */
  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:   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_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 current time */
  get_current_time(&timestamp);

  /* get node ID */
  get_ieee_node_identifier(&node);

  /* get saved state from NV storage */
  f = read_state(&clockseq, &last_time, &last_node);

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

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

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

  UNLOCK;
  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) {
  FILE * fd;
  static int inited = 0;

  /* only need to read state once per boot */
  if (!inited) {
      fd = fopen("state", "rb");
      if (!fd)
           return (0);
      fread(&st, sizeof(uuid_state), 1, fd);
      fclose(fd);
      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) {
  FILE * fd;
  static int inited = 0;
  static uuid_time_t next_save;

  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) {
      fd = fopen("state", "wb");
      fwrite(&st, sizeof(uuid_state), 1, fd);
      fclose(fd);
      /* 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 whenever.
  Compensate for the fact that real clock resolution is
  less than 100ns. */
void get_current_time(uuid_time_t * timestamp) {
    uuid_time_t               time_now;
    static uuid_time_t  time_last;
    static unsigned16   uuids_this_tick;
  static int                  inited = 0;

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

    while (1) {
        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;
           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,        /* resulting UUID */
  uuid_t nsid,          /* UUID to serve as context, so identical
                           names from different name spaces generate
                           different UUIDs */
  void * name,          /* the name from which to generate a UUID */
  int namelen           /* the length of the name */
) {
  MD5_CTX c;
  unsigned char hash[16];
  uuid_t net_nsid;      /* context UUID in network byte order */

  /* put name space ID 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(uuid_t));
  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]) {
    /* Construct a version 3 uuid with the (pseudo-)random number
     * plus a few constants. */

    memcpy(uuid, hash, sizeof(uuid_t));

  /* convert UUID to local byte order */
  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
       -1   u1 is lexically before u2
        0   u1 is equal to u2
        1   u1 is lexically after u2
    Note:   lexical ordering is not temporal ordering!
*/
int uuid_compare(uuid_t *u1, uuid_t *u2)
{
  int i;

#define CHECK(f1, f2) if (f1 != f2) return f1 < f2 ? -1 : 1;
  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;
};

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

/* change to point to where MD5 .h's live */
/* get MD5 sample implementation from RFC 1321 */
#include "global.h"
#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 acquire a system wide 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) {
  char seed[16];
  FILE * fd;
  static inited = 0;
  static uuid_node_t saved_node;

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

/* system dependent call to get the current system time.
   Returned as 100ns ticks since Oct 15, 1582, but resolution may be
   less than 100ns.
*/
#ifdef _WINDOWS_

void get_system_time(uuid_time_t *uuid_time) {
  ULARGE_INTEGER time;

  GetSystemTimeAsFileTime((FILETIME *)&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.
  */

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

  MD5Init(&c);
  /* memory usage stats */
  GlobalMemoryStatus(&r.m);
  /* random system stats */
  GetSystemInfo(&r.s);
  /* 100ns resolution (nominally) time of day */
  GetSystemTimeAsFileTime(&r.t);
  /* high resolution performance counter */
  QueryPerformanceCounter(&r.pc);
  /* milliseconds since last boot */
  r.tc = GetTickCount();
  r.l = MAX_COMPUTERNAME_LENGTH + 1;
  GetComputerName(r.hostname, &r.l );
  MD5Update(&c, &r, sizeof(randomness));
  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(char seed[16]) {
  MD5_CTX c;
  typedef struct {
      struct sysinfo s;
      struct timeval t;
      char hostname[257];
  } randomness;
  randomness r;

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

  /* 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_from_name(&u, NameSpace_DNS, "www.widgets.com", 15);
    printf("uuid_create_from_name() -> "); puid(u);
  };

  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");
  };
```

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_from_name()   -> e902893a-9d22-3c7e-a7b8-d6e313b71d9f
```

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 in section 3.5

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

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

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

uuid_t NameSpace_X500 = { /* 6ba7b814-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b814,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
  };
```