

THE UUID

A General Technology Paper



The UUID

UUIDs (and Microsoft Corporation's subset, the GUID), are values designed to be spatially and chronologically unique through out the universe, thus the term Universal Unique Identifier. This means that a UUID's uniqueness does not just span the distance of the universe, but the life time of the universe as well.

UUIDs are generally derived by running a tool, such as GuidGen.exe. However, the apparently randomly generated values from these tools have two limitations:

First, the values are not sequential. You cannot readily generate additional values in sequence to a previous one at a later date.

Second – and this is the really critical issue facing UUIDs today – the UUIDs are random to human perception. There is no way to search for a set of UUIDs based on any common fragment. For example, one cannot readily identify a set of UUIDs by a common subset of bits, as one might in the following pairs:

10000332-baa7-2812-c87a-0a9882773726

10000332-9832-b310-7543-ab30c8372bde

or...

9382b3ca-0192-DEAD-BABE-0192bae57c28

8372bed6-f73e-DEAD-BABE-b726d82ac638

The Makeup of a UUID

The UUID can be seen as a combined, four part integral value. The first part is a time stamp. This is literally a value of time with resolution of 0.1 milliseconds from the time the Gregorian calendar was put into effect. This implies (as it was intended) that it will continue to identify time through out the rest of the universe's life time. The second is a clock sequence, which is to avoid loss of monotonicity, to compensate for the limitation of the hardware clock (to avoid duplication of a value of time).

The third is an address, based upon a network node. The fourth is a UUID version, of which Microsoft holds a subset of $1/8^{\text{th}}$ all possible UUIDs as a GUID. In this sense, a GUID is a UUID, but one that is Microsoft generated.

This is the industry standard today. It is based upon today's computational and networking models. It is also limited by these models, in that to have a machine derived UUID value, many bits of resolution are sought to ensure statistical improbability of collision.

I say *statistical improbability*, because though some would say that a UUID, if produced properly, can never collide with another UUID, the problem is that one can not ensure

THE UUID

A General Technology Paper

that all UUIDs are produced *properly*. Thus, there is a possibility, and perhaps even a probability, that collision can occur regardless of all the safe guards one might take.



How Many Bits Are Really Necessary?

UUIDs have several qualities:

- 1) UUIDs are identifiers. To be successful identifiers, they must be unique. Their uniqueness – that is, the statistical certainty of their uniqueness – is their one valuable quality over other identifiers.
- 2) UUIDs are exactly 128 bits in size, regardless of the author of the UUID. In contrast, Java, for example, requires strings of various sizes to form unique identifications of packages, for those identifiers must be readily read and understood by programmers.
- 3) UUIDs are random in appearance to the untrained eye, and purely produced by machines without human understanding of the context for which they will be used.

The purpose of the UUID is to give a machine the ability to produce a unique identifier. This begs the question, are a total of 128 bits necessary to ensure uniqueness? To answer this question, we need to look further into how the statistical uniqueness is practical to the application at hand. And if 128 bits are not necessary, then some of the bits in a UUID can be allocated for other uses.

THE UUID

A General Technology Paper



64 bits is plenty!

Microsoft gave us GUID in place of UUID - global in place of universal. According to Microsoft, “Universal” is presumptuous for 128 bits. That is, given 128 bits, there is still a statistical probability or likeliness that a collision will occur within the universe over its lifetime. On the other hand, 128 bits is more than enough for the limits of the earth, which is where the term *global* comes from to form GUID.

This section will take a look at each component to examine what is required and what is fallacious.

The Time Component

Microsoft’s argument makes no sense in light of the lifetime of any UUID today. Can someone imagine a UUID identifying an interface in a Windows application continuing service 50 years from now? Here is a clue, folks. Bill Gates will be in his grave, Windows 2050 will not be called Windows 2050, and Win32 will be replaced with Win256 or WinQuantum. COM will be replaced with a more optimal and better thought out component architecture with identifying traits other than those criminally negligent CLSIDs (dear God, what were they thinking?), and the second coming will have occurred.

Instead of worrying about collision over the lifetime of the universe, or even the earth, one should focus on the lifetime of the UUID’s service – the lifetime of the entity it is charged to identify. And to properly do this, one must consider the technology it is deployed to support. Now we are beginning to see that it is an understatement to say that 128 bits is overkill.

In a UUID, 60 bits are used to represent time to 100 microseconds, and this is extended further with an additional 14 bits to reduce the probability of two time stamps being identical from the same machine’s clock. (This is just a limitation of the hardware.)

When a UUID identifies time at the resolution of 100 microseconds resolution, a century’s worth will consume 42 bits. Given a UUID is published every 100 microseconds, one can expect a collision within 100 years when using only 42 bits.

If we say that the service of a UUID is well below 50 years (this is self evident, folks), then a UUID can be produced every 100 microseconds without producing a collision in the field (in a consumer’s PC) using only 42 bits.

Since a UUID is published by a single software engineer on the scale of perhaps one a day (let’s give them the benefit of the doubt, though we strrrrrrrretch here), one can readily see that 16 bits is more than enough to maintain uniqueness over the lifetime of the UUID’s service for a given publisher. If 16 bits offers the resolution required to identify every UUID for a single publisher, then 32 bits of machine generated random numbers should statistically ensure uniqueness without any relationship to time apart from seeding the RNG.

THE UUID

A General Technology Paper

Thus nearly 80 bits of time stamp and correction can be replaced with 32 bits of machine generated random value in its place and produce adequate protection against collision for a single publisher.



The Spatial Component

48 bits are used to identify the spatial component of the UUID, to ensure uniqueness over the distance of the universe. But in practice, it is based upon the network node's MAC. This means that the quantities of values that can be produced are directly related to the number of MAC addresses that can be humanly published. This is directly related to the physical manufacturing process of hardware, which is extremely limited. The only way to break past this limitation is to virtually manufacture network nodes and assign them MACs. But this is not an industry practice of real concern. The only significant source of virtual MACs is found in virtual machines, such as those produced by VMware, Inc.

A safe world wide ceiling of total MACs across all networks would be 4G. To identify this many nodes would require 32 bits. Yet, the industry standard uses 48 bits to form a MAC, for a total of 256T nodes.

A Dramatically Hypothetical Case

Given the upper bounds hypothetical case of 1M processors across the earth each producing 10G UUIDs every year over a period of 100 years, the total yield is 10^{18} (2^{60}) UUIDs. With 60 random bits, we should not expect to see a collision for 50 years.

Now let's venture back into reality for a moment.

1. We will never see 10,000,000,000 UUIDs in our life time, let alone in a single year. In fact, none of us will ever see 10,000,000 UUIDs in our life time, and none of use will see 100,000 UUIDs in a single year.
2. A good estimate of the number of PCs that derive UUIDs might be 1,000,000, but even then we are talking about a few published UUIDs each year on average.
3. The life expectancy for the average UUID is far less than 100 years. That being the case, many UUIDs can be recycled - reused without collision.
4. UUIDs will be extinct within 100 years, so recycling does not enter into the equation either, though it can occur by chance.

If we make a realistic expectation over the next 100 years that we expect to see no more than 1,000,000,000 UUIDs, and most of these will never see service for more than twenty years, then 32 bits of randomly generated values would be quite adequate to protect against collision during this time.

Conclusion

Assuming a ceiling for an identifier's service life of 100 years, using no more than 32 bits of random number generation by a machine seeded by the clock and MAC can produce adequate uniqueness to avoid collision well beyond the service life of any identifier. The

THE UUID

A General Technology Paper

only aspect of the UUID's time component that is lost is the vast future dating, which is useless since UUIDs will be extinct within 100 years.





Introducing XUID

There is no need for 128 bits of unique values that remain unique for thousands of millions of years. No one can take such thinking seriously. Yet, that is precisely what the industry has chosen to do.

48 bits of random number is more than adequate as long as it has a large repeat period. One RNG to use may be the *Mother Of All RNGs*, documented by Dr. Michel Rosing in his book, Implementing Elliptic Curve Cryptography, created by Dr. George Marsaglia. This RNG has a reported cycle of 2^{250} numbers, perhaps making it the strongest RNG known to man today. But while the period is impressive, it may perhaps be quite useless for our needs.

It must be pointed out that an RNG can only have a period greater than the number of seed bits if it has the opportunity to run past the first repeat point. Otherwise, reseeding will result in a restart of the initial seeding and a repeat seeding will never occur. This limitation becomes unavoidable when the PC that generates the ID is rebooted and the internal state of the RNG is not persisted.

What is ideal is an RNG that takes a seeding value with each pass to incorporate along with its internal state, thus giving each pass fresh time input down to milliseconds of volatile randomness, and previous internal state to compensate for granular time resolution (like the clock sequence in the UUID is used for).

Enter the XUID based upon seeding an RNG with the time and MAC, producing 48 bits that are expected unique, consuming 128 bits of a UUID, the XUID provides 80 bits for use other than uniqueness in identification, though they can also be used in a dual use for identification.

For example, the following values can all be presumed strong, unique identifiers:

00000000-0000-0000-0000-a01182bd6349 make the first 80 bits all zero and give Bill Gates a heart attack by stacking your UUIDs at the top of his registry

00000100-0200-0000-4000-a713b2834b50 encodes special values that group identifiers by product, functionality, region, etc.

Another Level Of Reality

Now let's discuss the need for randomness period. In Java, a company, BriansEnterprises, can use its name along with the package and filename as the fully qualified namespace for the class: **com.briansenterprises.packagename.file.java**.

Why the name of the company? Because by abiding by that name, it is extremely unlikely anyone else would use the same name. But it doesn't prevent such a collision.

Take that to a numeric equivalent, BriansEnterprises could have been a numeric expression: **188827462120-8332-00a7-b766-xxxxxxxxxxxxx**, where x is a random



THE UUID

A General Technology Paper

component and 188827462120-8332-00a7-b766 identifies the business. What are the odds that anyone else would use that same value for business identification? This would allow Brian's Enterprises to compete for the other 48 bits against only himself. (But even if someone else used those same 80 bits, the 48 unique bites guarantee a statistical improbability of collision.)

With 80 bits of entity identification, one can begin to see how the UUID value **00000001-0000-0000-2222-00000000192** is the 192nd identifier assigned by the entity, and note how easy it is to identify the entity: 1-0-0-2222. The shorthand for the UUID is then 1-0-0-2222-192.

Let's take a more interesting case. Let's say a business identifies itself numerically by using its main telephone number. For example, a business whose main number is 808-329-3300 may use a format like: 00000000-**0808-0329-3300**-xxxxxxxxxxx. The short hand would be 0-808-329-3300-xxxxxxxxxxx.

Given that nearly the entire industry will brush this approach off as "unethical", one can use UUIDs like 0-0-0-819-*id* with absolutely no concern for collisions! In this case, one would rely on the industries "sheepishness" to follow the "authorities" to avoid collision for them. The best part is that all XUIDs can be grouped to list at the top or the bottom of a Windows registry CLSID or Interface list!

THE UUID

A General Technology Paper



Conclusion

The industry has chosen a course of action to standardize the creation of identifiers across the universe and across all time. What they did in effect was substitute elegance for common sense.

With the introduction of COM, Microsoft made identifying components very painful. With Java and now .NET, this trend has been reversed to make identification of components quite easy and user friendly. This trend is expected to continue and UUIDs should quickly be phased out over the next 50 years as Win32 is replaced with a new generation platform.

In the mean time, an identifier can safely be generated by a strong random generator protocol with just 48 bits and statistically ensure no collisions well beyond the service life of any identifier. Backward compatibility is maintained with existing UUIDs by using only a small part of the UUID for collision avoidance. Even with the larger portion being static, the complete value as a whole helps identify and avoid collision with all other UUIDs.

Using XUIDs can make identifying COM components quite easy once again, though perhaps not nearly as easy as assembly or package naming conventions would yield.