UID v1.0

*by **digi_cs** Jul-Sep 2024* *cosmogen@gmail.com*

Where am I taking you?
To the Andromeda Galaxy .

This library provides a service for generating UIDs (Unique IDs) – unique strings with a controlled structure.

Unique strings in AWK can have many applications, with the most important being their **use as pointers**.

Pointers are the **most powerful method of data addressing** in AWK.

Each pointer can address an **unlimited number of arrays and strings** stored in global array-fields by indexing its string:

```
ARRAY[ ptr ]
```

The use of pointers elevates the AWK language to a completely new level, unlocking **new dimensions and possibilities** in this programming language. In fact, it introduces a new data type that AWK can handle: pointers.

See the section **POINTERS IN AWK**.

Additionally, this library includes an implementation of HID (Hidden ID) — an essential resource for AWK programming, particularly (but not exclusively) in working with arrays (refer to **HIDs IN AWK** for more details).

The library has been tested and is fully functional for both regular and byte (-b) modes of gawk.

USE

To use the library, execute the following:

```
@include "uid.lib"
```

The library features **dynamic initialization**. This means that the library's resources can be called from **any point in the code** (including from the `BEGIN{}` sections of files included in the main source before the library itself is included).

The library will automatically initialize upon the first invocation of any of its functions, but no later than the execution of the library's own `BEGIN{}` section.

UID

Unique strings (UIDs) are generated using declared UID generators.

When declaring a UID generator, parameters are passed to it, which define the content of the strings generated by this generator (i.e., what they will be composed of).

UID generators are addressed via pointers — any string can serve as a pointer.

The general structure of generated unique strings is shown below:

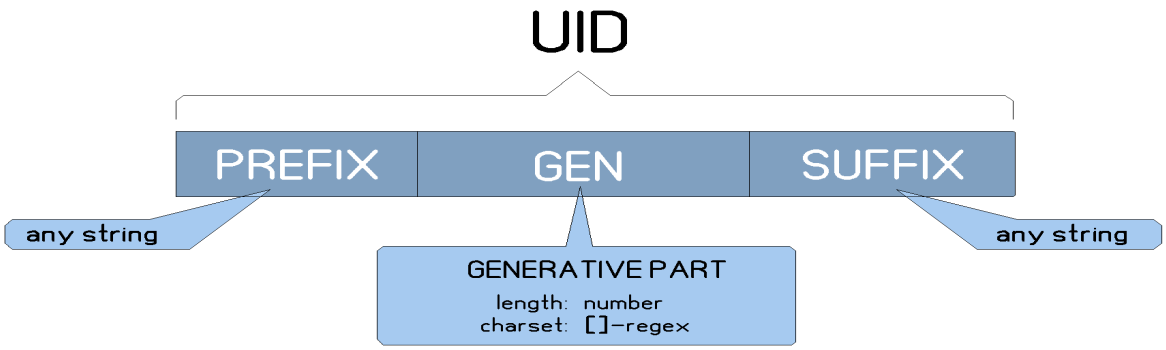


Diagram 1

A prefix and suffix are added to the left and right of the generative part of the UID, which can be of any length and consists of characters specified by the `charset` parameter.

To generate a unique string using a previously declared UID generator, the following function is called:

`uid::get``( ptr )`

where:

- `ptr` – a pointer to the UID generator that will be used to generate the string.

PERFORMANCE: 5.7 MUID/sec for default uid-generator on i7/2GHz.

To declare an **independent UID generator**, the following function is called:

uid::set(*ptr*, *prefix*, *suffix*, *length*, *charset*)

example:

```
ptr = "Gen1"

uid::set( ptr, "<", ">", 3, "01" )

...
```

A UID generator is declared with the pointer **"Gen1"**, having the prefix "**<**", the suffix "**>**", and a generative part consisting of **three** characters from the set: **"0"** and **"1"**.

... and now we start generating strings using the declared UID generator:

```
...

while ( 1 )

      print ++CNT ":\t" uid::get( ptr )

outputs:

      1:    <000>
      2:    <001>
      3:    <010>
      ...
      7:    <110>
      8:    <111>

      fatal: uid::get( Gen1 ): out of uid range
```

The characters in the generative part of the UID are iterated incrementally, similar to how digits in numbers are incremented.

A fatal error occurs when all possible combinations of characters are exhausted.

> Parameter *charset* is provided as a string: a `[]`-fragment of a regular expression that corresponds to the characters allowed for use in the generative part of the UID.

The selection of characters (chosen by the regular expression) is limited to characters in the range `\x00-\xFF`.

> Parameter *charset* may be omitted (== `""`). In this case, the default value will be `"\xC0-\xEF"`, which corresponds to 48 characters with hex codes from C0 to EF.

> Parameter *length* defines the number of characters in the generative part of the UID generator. It can be any value >= `1`.

> Parameter *length* may be omitted (== `""`). In this case, the default value will be `4`.

UID-GENERATOR

The **character set** is represented as a list-array, sequentially enumerating all possible combinations that the given character set can return.

Depending on the requirements, the charset list-array may contain either:

- A list of all possible single characters (e.g., for an odd *`length`*)
- All combinations of two such possible characters (e.g., for *`length`* `> 1`).

Below is an example of a single and a double charset list-array:

```
single character set list array for characters: "A-C"

      [ "" ] = "A"
      [ "A" ] = "B"
      [ "B" ] = "C"
      [ "C" ] = - (undefined)

double character set list array for characters: "01"

      [ "" ] = "00"
      [ "00" ] = "01"
      [ "01" ] = "10"
      [ "10" ] = "11"
      [ "11" ] = -
```

The UID generator has two character sets: `low` and `high`, and two counters: `_COUNTL` and `_COUNTH`, respectively, which store the latest values obtained from the character sets.

This allows for the iteration of 1-4 characters in the generative part of the UID.

> If the *`length`* exceeds 4, "stacked" UID generators are used — these do not have their own prefix or suffix but essentially replicate the structure of the "declared" UID generators.

Depending on the *`length`*, the required number of these "stacked" UID generators is created at a rate of one UID generator for every additional 1-4 characters in the generative part of the UID.

Below is the structure of the UID generator:

UID-GENERATOR STRUCTURE

```
                    namespace uid

  _PREFIX[ ptr ]                    = "prefix"
  _SUFFIX[ ptr ]                    = "suffix"
_COUNTPTR[ ptr ]                    = countptr
_CHARSETL[ ptr ]                    = charsetptr
_CHARSETH[ ptr ]                    = charsetptr

  _COUNTL[ countptr ]               = last returned low-part
  _COUNTH[ countptr ]               = last returned hi-part

   _NEXT[ countptr ]                = ptr to next uid-generator
  _HIPFX[ countptr ]                = last uid returned from the
                                    next uid-generator

_CHARSET[ charsetptr ][ ... ]       character set list-array
```

Note that the fields _COUNTL, _COUNTH, _NEXT, and _HIPFX are not located by the pointer *ptr*, but by the pointer *countptr*, which is stored in the _COUNTPTR[ptr] field.

Similarly, the low and high character sets are not tied to ptr; they are linked to pointers *charsetptr*, stored in the _CHARSETL[ptr] and _CHARSETH[ptr] fields, respectively.

From this, we can understand an important point: the counters _COUNTL/_COUNTH and the low/high character sets of a UID generator can be "remote" — i.e., shared (from another UID generator).

This brings up the idea of two additional modes in which a UID generator can be initialized beyond the basic (independent) mode.

The first mode is when the UID generator uses remote (shared) character sets, but has its own prefix, suffix, and most importantly, its own counters for iterating through characters in the character set (i.e., it keeps its own count of combinations in the generative part of the strings, but uses someone else's generative part as a template: the same length and from the same characters).

Such a UID generator will generate unique strings with its own prefix and suffix, but with a generative part identical to the remote UID generator from which it uses the character set, while maintaining its own iteration sequence in the generative part.

To set up a UID generator in this mode, known as the **remote character set mode**, the following function is used:

uid::setcs(*ptr*, *prefix*, *suffix*, *charsetptr*)

example:

```
ptr1 = "Gen1"
ptr2 = "Gen2"

uid::set( ptr1    , "<", ">", 3, "01" )

uid::setcs( ptr2  , "[", "]", ptr1 )

…
```

A UID generator is declared with the pointer "Gen1", having the prefix "<", the suffix ">", and a generative part consisting of three characters from the set: "0" and "1".

A UID generator is declared with the pointer "**Gen2**", having the prefix "**[**", the suffix "**]**", and a generative part identical to that of the UID generator "Gen1" (i.e., it has the same length (**3**) and consists of the same characters: **0** and **1**).

... and now we start generating strings using both declared UID generators:

```
...

while ( 1 ) {

      print ++CNT ":\t" ptr1 "\t" uid::get( ptr1 )

      print   CNT ":\t" ptr2 "\t" uid::get( ptr2 ) }
```

outputs:

```
1:    Gen1  <000>
1:    Gen2  [000]
2:    Gen1  <001>
2:    Gen2  [001]
...
7:    Gen1  <110>
7:    Gen2  [110]
8:    Gen1  <111>
8:    Gen2  [111]

fatal: uid::get( Gen1 ): out of uid range
```

> The parameter *charsetptr* may be omitted (== ""), in which case a pointer to the **default UID generator** will be used.

Another possible mode of the UID generator is when both the character set and the counters `_COUNTL`/`_COUNTH` are remote (i.e., not its own, but from another UID generator).

In this mode, the UID generator will generate unique strings with its own prefix and suffix, but will have a generative part generated by another UID generator.

To set up a UID generator in this mode, known as the **shared character set mode**, the following function is used:

`uid::setcnt``( ptr, prefix, suffix, countcharsetptr )`

```
example:

      ptr1 = "Gen1"
      ptr3 = "Gen3"

      uid::set( ptr1    , "<", ">", 3, "01" )

      uid::setcnt( ptr3 , "{", "}", ptr1 )

      ...
```

A UID generator is declared with the pointer `"Gen1"`, having the prefix "<", the suffix ">", and a generative part consisting of three characters from the set: `"0"` and `"1"`.

A UID generator is declared with the pointer **`"Gen3"`**, having the prefix **`"{`**", the suffix **`"}`**", and the generative part from the UID generator `"Gen1"` (i.e., it uses the exact generative part, not an equivalent).

... and now we start generating strings using both declared UID generators:

```
      ...

      while ( 1 ) {

            print ++CNT ":\t" ptr1 "\t" uid::get( ptr1 )

            print ++CNT ":\t" ptr3 "\t" uid::get( ptr3 ) }

      outputs:

            1:    Gen1  <000>
            2:    Gen3  {001}
            3:    Gen1  <010>
            4:    Gen3  {011}
            5:    Gen1  <100>
            6:    Gen3  {101}
            7:    Gen1  <110>
            8:    Gen3  {111}

            fatal: uid::get( Gen1 ): out of uid range
```

> The parameter *countcharsetptr* may be omitted (== ""), in which case a pointer to the default UID generator will be used.

When setting up UID generators in the remote or shared character set modes, you can reference not only independently declared UID generators but also those operating in the remote or shared character set modes.

PREDEFINED UID-GENERATORS

The library provides two predefined UID generators: the default generator and the HID generator.

Both predefined UID generators cannot be canceled or redeclared.

All other UID generators can be freely redeclared in any mode.

DEFAULT UID-GENERATOR

Pointer: **""** (empty string)

The default UID generator generates strings with the following structure:

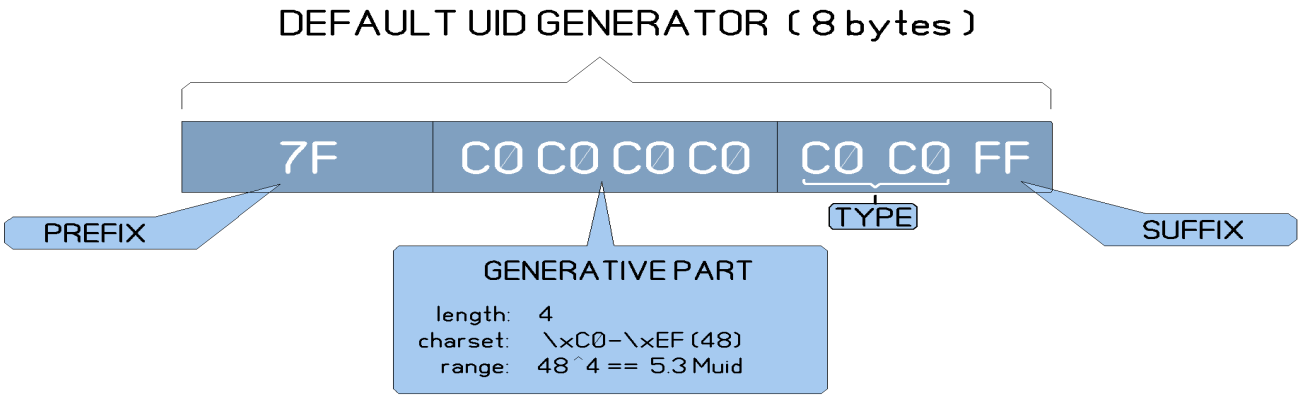


Diagram 2

```
uid::set( "", "\x7F", "\xC0\xC0\xFF", 4, "\xC0-\xEF" )
```

The default UID generator is essential for the library's proper functioning and cannot be overridden or redeclared.

Refer to the **POINTERS IN AWK** section for more information.

HID-GENERATOR

Pointer: “hid”

The HID generator produces strings with the following structure:

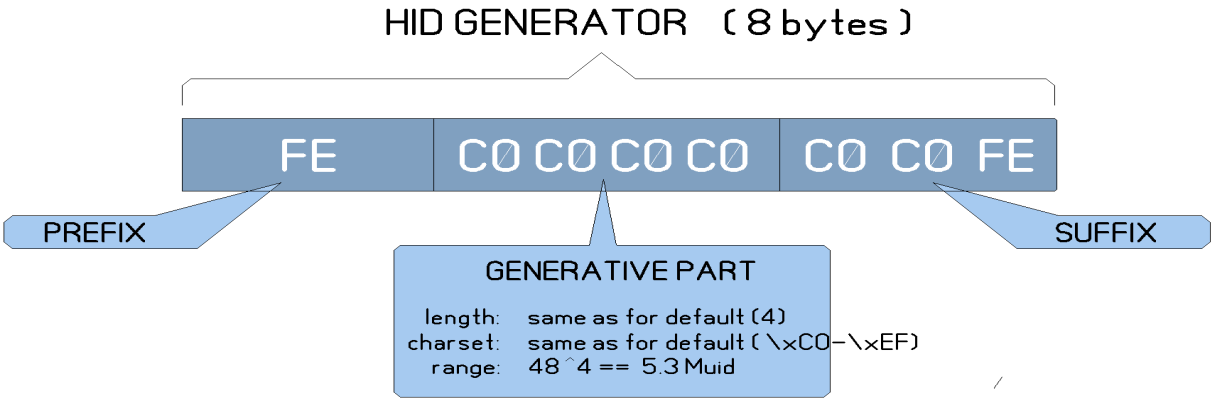


Diagram 3

```
uid::setcs( "hid", "\xFE", "\xC0\xC0\xFE" )
```

HIDs IN AWK

HID is a **very important resource** in AWK programming.

Whenever you're about to **store** external **data** in an **array**, you are repeatedly faced with the same question: **under what indices** should you do it?

Indeed, the array contains some "**working**" **indices**, and it is critical that the "service" indices **do not conflict** with any possible "working" indices.

And if we simply assign the "service" indices as a **direct string**, for example:

```
A["LIST"]
```

...then for obvious reasons, we will quickly run into such a conflict.

Code readability is also important: if you start complicating the array index string with additional characters to minimize the chance of conflicts with **working indices**, you will inevitably reduce its readability:

```
A[ "\xFELIST\xFE" ]
```

This creates a **paradox**: we want a **perfectly clear index** notation, but such notation may **conflict with working indices**. On the other hand, if we **complicate the index**, we inevitably **lose comfort** in usage.

HID is a system-level solution to this problem.

HID is a global variable, referred to as a "carrier," which is **declared using the `hid()` function** and is assigned a so-called **strong value** generated by the **HID generator**.

The essence of the value assigned to the HID variable is that it is designed to be unlike anything else—neither potential data nor possible "working" indices in arrays.

Users can actively use carrier HID variables as a sort of "service" index in arrays, allowing access to additional, auxiliary information stored within the array:

`A[ SOME_ ]` instead of: `A[ "SOME_" ]`

IMPORTANT! A negative factor when using `hid` is the fact that if the `hid` variable is not declared or its name is misspelled, the user will not be directly notified about it. The code will start executing unpredictable actions (an undeclared variable will return an empty string instead of a specific value).

Therefore, special attention should be paid to writing `hid` in the code and ensuring that all `hid` variables being used are declared beforehand.

Some service indices we will need are: - local ones, i.e., internal to the specified namespace. Others must be declared with the same value across any namespace (global).

`hid` declarations must be fully made in each namespace where they are used.

The so-called superglobal `hid`s — whose carrier variables are superglobal variables — must also be declared in every namespace where they are used.

The declaration of `hid` is done by assigning the carrier `hid` variable the result returned by the `hid()` function, called with the name of the declared `hid` as a string parameter.

hid::get(*name*, *code*)

example:

```
SOME_ = hid::get( "SOME_" )
print hexdump( SOME_ )
```

outputs (as example):

```
FE C0 C1 C2 C3 C4 C5 FE
```

The `hid` is declared with the name: *name*. The carrier `hid` variable *name* is assigned a specific special value.

The parameter *code* allows the user to declare a `hid` using their own special value.

> If *code* `!= ""`, it becomes the new special value. Otherwise, a new special value will be generated by the `hid` generator.

In some cases, the user simply needs a special value but does not require its registration as a `hid`:

> If the parameter *name* is an empty string (== ""), then the `hid()` function simply returns a new special value:

```
... = hid::get(
```

The parameter *name* contains the name of the declared hid. The name can be provided with or without a namespace. The namespace separator can be either '::' or '.'

> If the `hid` *name* has not been declared up to this point, the `hid()` function will return a new **special value** and associate it with *name*.

In subsequent attempts to declare the same *name*, the function `hid::get()` will simply return the stored value.

> If *name* does not contain a namespace separator, a global `hid` is declared:

```
GLOBAL_     = hid::get( "GLOBAL_" )
```

A global `hid` carries the same value in any namespace.

```
@namespace = "one"
BEGIN{
      GLOBAL_ = hid::get( "GLOBAL_" )
@namespace "two"
BEGIN{
      GLOBAL_ = hid::get( "GLOBAL_" )
# one::GLOBAL_ == two::GLOBAL_
```

If the parameter *name* contains a **namespace separator**, a local `hid` is declared:

```
@namespace "some"

BEGIN{

      LOCAL_ = hid::get( "some::LOCAL_" )
```

A local `hid` is declared only for its own namespace and always carries its own value.

> If the parameter *name* contains a namespace separator and does not specify a namespace, the namespace that was last specified for the `hid::get()` function is used:

```
@namespace "some"

BEGIN{

      LOCAL1_ = hid::get( "some.LOCAL_" )

      LOCAL2_ = hid::get( ".LOCAL2_" )
```

HID[*code*]

To determine whether a certain string is one of the "hid", the user can check the existence this string as an index in the **superglobal array `HID`**.

For example, within a `for-in` loop:

```
for ( i in A )

      if ( i in HID )

            continue

      else ...
```

The superglobal array **`HID`** contains all special values returned by the hid::get() function as its indices.

hid::name(*code*)

The function returns the **name of the `hid`** associated with the code *`code`*.

The parameter *`code`* must be the value of one of the declared carrier `hid` variables.

If *`code`* is not such a value, the function returns *null*.

```
example:
      BEGIN{
            GLOBAL_           = hid::get( "GLOBAL_" )
            some::LOCAL1_     = hid::get( "some::LOCAL1_" )
            LOCAL2_           = hid::get( ".LOCAL2_" )

            print hid::name( GLOBAL_ ) "'"
            print hid::name( some::LOCAL1_ ) "'"
            print hid::name( LOCAL2_ ) "'" }
outputs:
      GLOBAL_'
      some::LOCAL1_'
      some::LOCAL2_'
```

POINTERS IN AWK

For a better understanding of how to use pointers in AWK, it is recommended to first familiarize yourself with the source code of the library itself:

- You will see how to properly arrange data in relation to the pointer:

  `FIELD[ ptr ]`

  For example, a certain pointer `ptr` has the mandatory property: ID located in:

  `ID[ ptr ]`

- How to determine if `ptr` is a pointer:

  To check if the string `ptr` is an active pointer, you need to execute:

```
if ( ptr in ID )
      # ptr is pointer
else  # ptr is something else
```

The bodies of pointers have a characteristic structure that can be easily recognized using regular expressions among the data: each pointer starts with the character \x7F and then, after 6 bytes, ends with the character \xFF.

Thus, "something else" in the example above can mean that the string *ptr* is any other string consisting of possible data and pointers.

The essence of the proposed methodology for working with data is that the original string for us is a mixture of data and pointers.

To search for pointers of any type, it is suggested to use the following regular expression:

```
/\x7F......\xFF/
```

If you need to search for pointers of a specific type(s), you should specify the possible characters in the type field of the pointer being searched (the two penultimate bytes in the pointer — see Diagram 2):

```
/\x7F....\xT0\xT1\xFF/
```

where: **\xT0** and **\xT1** – type (class) identifier characters of the pointer being searched.

To avoid inadvertently matching the bodies of pointers among the data with any external regular expressions, you should avoid the character \x7F (when moving right) and the character \xFF (when moving left).

FYI

UID generators in my environment are also used for generating temporary file names.

I strongly advise against experimenting with the settings of the default and HID generators: trust me, the structures of the strings they generate are not arbitrary or random. For me personally, they are third-generation pointers.

PROSPECTS

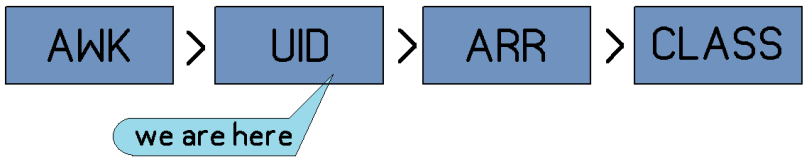
At the first stage, we need to raise the overall level of basic programming techniques in AWK.

This library pertains to the basic programming techniques in AWK.

The next release will be the ARR library—for working with arrays (also basic techniques).

Then, the release of the CLASS library will follow—representing a class implementation for AWK: the fundamentals of basic techniques and a description of OOP practices in AWK.

A hypothetical publication plan for AWK libraries for the year 2025:



AUTHOR

Class implementation in AWK is a key factor for the further comprehensive development of this language.

My personal "Andromeda Galaxy" lies in working with complex regular expressions, and that is exactly where I'm leading you.

Thank you for your interest and attention!

Kind Regards
Denis Shirokov

RESPECT DUE

All awkers from all the world!

gawk Team