

# UID v1.0

by **digi\_cs** Jul-Oct 2024

cosmogen@gmail.com

*Куда я вас веду?  
В галактику Андромеды.*

Данная библиотека предоставляет сервис для генерации UID ( Unidue ID ) - уникальных строк с контролируемой структурой.

Уникальные строки в AWK могут иметь множество применений из которых важнейшим является их **использование в качестве указателей**.

Указатели являются **наиболее мощным способом адресации** данных в AWK..

Каждый указатель может адресовать **неограниченное количество массивов и строк** располагаемых в глобальных массивах-полях по индексу его строки:

```
ARRAY[ ptr ]
```

Применение указателей выводят язык AWK на совершенно новый качественный уровень открывая в этом языке программирования **новые измерения и возможности**. Фактически можно говорить о новом типе данных с которым умеет работать AWK: **указателях**.

Смотрите раздел **POINTERS IN AWK**.

Также данная библиотека включает в себя реализацию **HID** ( Hidden ID ) — совершенно необходимого в программировании на AWK ресурса особенно (но не только) в части работы с массивами (смотрите раздел **HIDs IN AWK** ).

Библиотека протестирована и полностью работоспособна для обычного и байтового (-b) режимов gawk.

## USE

Для использования выполните:

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

В библиотеке реализована **динамическая инициализация**. Это означает что ресурсы библиотеки можно вызывать **из любых точек кода** (в т.ч. и из BEGIN{}-зон файлов включаемых в общий исходник ранее — до инклюда данной библиотеки).

Инициализация библиотеки **произойдёт автоматически** в момент первого вызова любой из её функций, но не позднее выполнения BEGIN{}-зоны данной библиотеки.

## UID

Уникальные строки (**uid**) генерируются с помощью объявляемых **uid-генераторов**.

При объявлении uid-генератора - ему передаются параметры определяющие содержимое генерируемых этим генератором строк (т. е. определяющие то, из чего они будут состоять).

Адресуются uid-генераторы с помощью указателей — любой строки.

Общая структура генерируемых уникальных строк показана ниже:

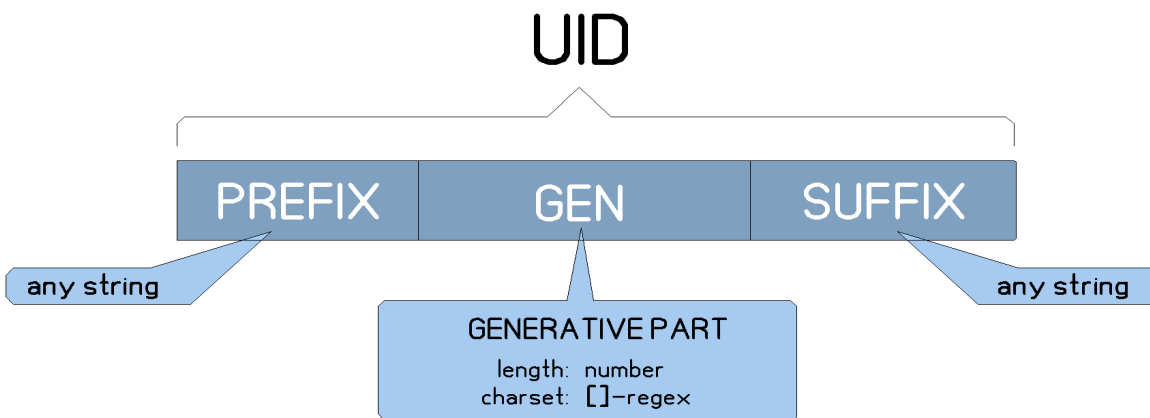


Diagram 1

Префикс и суффикс добавляются соответственно слева и справа к генеративной части uid которая может быть любой длины (length) и состоять из перебора символов указываемых параметром charset.

Чтобы сгенерировать уникальную строку с помощью некоторого объявленного ранее uid-генератора вызывается функция:

```
uid::get( ptr )
```

где: ptr — указатель на uid-генератор с помощью которого производится генерация строки

PERFORMANCE:                    **5.7 Muid/sec for i7/2Ghz**

Для того чтобы **объявить независимый uid-генератор** вызывается функция:

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

example:

```
ptr = "Gen1"

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

...
```

Объявляется uid-генератор с указателем "Gen1" имеющий префикс "<", суффикс ">" и генеративную часть из **трёх** символов из числа: "0" и "1".

... и теперь начинаем генерировать с помощью объявленного uid-генератора строки:

```
...

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
```

Символы в генеративной части uid перебираются инкрементально также как это делают цифровые разряды в числах.

При выборе всех возможных комбинаций символов возникает фатальная ошибка.

> Параметр *charset* подаётся в виде строки: []-фрагмента регулярного выражения соответствующего тем символам которые разрешены для применения в генеративной части uid. При этом выбор символов (из которых выбирает данное регулярное выражение) ограничен символами в диапазоне: \x00-\xFF.

> Параметр *charset* может быть не указан ( == "" ). в этом случае значением по-дефолту станет: "\xC0-\xEF"- что соответствует 48ми символам с hex-кодом: от C0 до EF.

> Параметр *length* определяет количество символов в генеративной части uid-генератора. **оно может быть любым** >= 1.

> Параметр *length* может быть не указан ( == "" ). в этом случае значением по-дефолту станет: 4

## UID-GENERATOR

Character set представляет собою список-массив перечисляющий последовательно все возможные комбинации которые данный character set может вернуть.

В зависимости от необходимости charset список-массив может содержать в себе как список всех возможных **одиначных** символов ( например при нечётном length ), так и все комбинации **двух** таких-же возможных символов ( например при length > 1 )

Ниже приведён пример одиночного и двойного charset список-массива:

```
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" ] = -
```

Uid-генератор имеет **два character set**: low и high и два счётчика `_COUNTL` и `_COUNTH` соответственно - содержащие последние значения полученные от character set

Таким образом осуществляется перебор 1-4 символов для генеративной части uid.

Если length превышает значение 4, то используются «надстроенные» uid-генераторы — не имеющие собственного префикса и суффикса, но по сути — повторяющие структуру тех-же «объявляемых» uid-генераторов.

В зависимости от length создаётся нужное количество таких “надстраиваемых» uid-генераторов из расчёта: один uid-генератор на каждые следующие 1-4 символа в генеративной части uid.

Ниже показана структура uid-генератора:

## UID-GENERATOR STRUCTURE

namespace uid

<code>_PREFIX[ ptr ]</code>	<code>= "prefix"</code>
<code>_SUFFIX[ ptr ]</code>	<code>= "suffix"</code>
<code>_COUNTPTR[ ptr ]</code>	<code>= countptr</code>
<code>_CHARSETL[ ptr ]</code>	<code>= charsetptr</code>
<code>_CHARSETH[ ptr ]</code>	<code>= charsetptr</code>
<code>_COUNTL[ countptr ]</code>	<code>= last returned low-part</code>
<code>_COUNTH[ countptr ]</code>	<code>= last returned hi-part</code>
<code>_NEXT[ countptr ]</code>	<code>= ptr to next uid-generator</code>
<code>_HIPFX[ countptr ]</code>	<code>= last uid returned from the next uid-generator</code>
<code>_CHARSET[ charsetptr ][ ... ]</code>	<code>character set list-array</code>

Обратите внимание что поля `_COUNTL`, `_COUNTH`, `_NEXT` и `_HIPFX` находятся не по указателю `ptr`, а по указателю `countptr` который хранится в поле `_COUNTPTR[ ptr ]`.

И также младший и старший `character set`: они не привязаны к `ptr`, они привязаны к указателям `charsetptr` хранящихся в полях `_CHARSETL[ ptr ]` и `_CHARSETH[ ptr ]` соответственно.

Из этого можно вывести для понимания тот важный момент что и счётчики `_COUNTL/_COUNTH` и младший/старший `character set` могут быть у `uid`-генератора удалёнными — т. е. чьими-то ещё (от какого-то другого `uid`-генератора).

И этот факт выводит на идею о двух дополнительных режимах в которых помимо основного (независимого) может быть синициализирован `uid`-генератор.

Первый режим это когда `uid`-генератор использует удалённые ( чьи-то ) `character set`, но при этом имеет свои собственные префикс и суффикс, а главное - собственные счётчики перебора символов в `character set` (т.е. ведёт собственный отсчёт комбинаций в генеративной части генерируемых строк, но при этом использует чужую генеративную часть как образец: та-же длина и из тех-же символов)

Такой `uid`-генератор будет генерировать уникальные строки с собственным префиксом и суффиксом и иметь генеративную часть полностью аналогичную удалённому `uid`-генератору от которого используется `character set`, но при этом имея собственную последовательность перебора символов в генеративной части

Для установки `uid`-генератора в этом режиме называемом режимом **удалённого character set**, используется функция:

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

example:

```
ptr1 = "Gen1"  
ptr2 = "Gen2"  
  
uid::set( ptr1      , "<", ">", 3, "01" )  
  
uid::setcs( ptr2   , "[", "]", ptr1 )  
  
...
```

Объявляется uid-генератор с указателем "Gen1" имеющий префикс "<", суффикс ">" и генеративную часть из трёх символов из числа: "0" и "1".

Объявляется uid-генератор с указателем "**Gen2**" имеющий префикс "[" суффикс "]" и генеративную часть аналогичную генеративной части uid-генератора "Gen1" (т.е. имеющую ту-же длину (**3**) и состоящая из тех-же символов: **0** и **1** ).

... и теперь начинаем генерировать с помощью обоих объявленных uid-генераторов строки:

```
...  
  
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
```

> Параметр *charsetptr* может быть не указан ( == "" ) и в этом случае будет взят указатель на дефолтовый uid-генератор.

Ещё одним из возможных режимов uid-генератора является режим когда и character set и счётчики `_COUNTL/_COUNTH` uid-генератора удалённые (т. е. не свои, а от какого-то другого uid-генератора)

Такой uid-генератор будет генерировать уникальные строки с собственным префиксом и суффиксом, но при этом будет иметь генеративную часть генерируемую другим uid-генератором.

Для установки uid-генератора в этом режиме называемом режимом **разделённого character set** используется функция:

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

example:

```
ptr1 = "Gen1"  
ptr3 = "Gen3"  
  
uid::set( ptr1 , "<", ">", 3, "01" )  
  
uid::setcnt( ptr3 , "{", "}", ptr1 )  
  
...
```

Объявляется uid-генератор с указателем "Gen1" имеющий префикс "<", суффикс ">" и генеративную часть из трёх символов из числа: "0" и "1".

Объявляется uid-генератор с указателем "**Gen3**" имеющий префикс "{" суффикс "}" и генеративную часть от uid-генератора "Gen1" (т. е. именно её, а не её аналог)

... и теперь начинаем генерировать с помощью обоих объявленных uid-генераторов строки:

```
...  
  
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
```

> Параметр *countcharsetptr* может быть не указан ( `== ""` ) и в этом случае будет взят указатель на **дефолтовый uid-генератор**.

При установке uid-генераторов в режимах удалённого или разделённого character set можно ссылаться не только на независимо объявленные uid\_генераторы, но и на такие-же uid-генераторы — работающие в режимах удалённого либо разделённого character set.

## PREDEFINED UID-GENERATORS

Библиотека предоставляет два предустановленных uid-генератора: дефолтовый и hid-генератор.

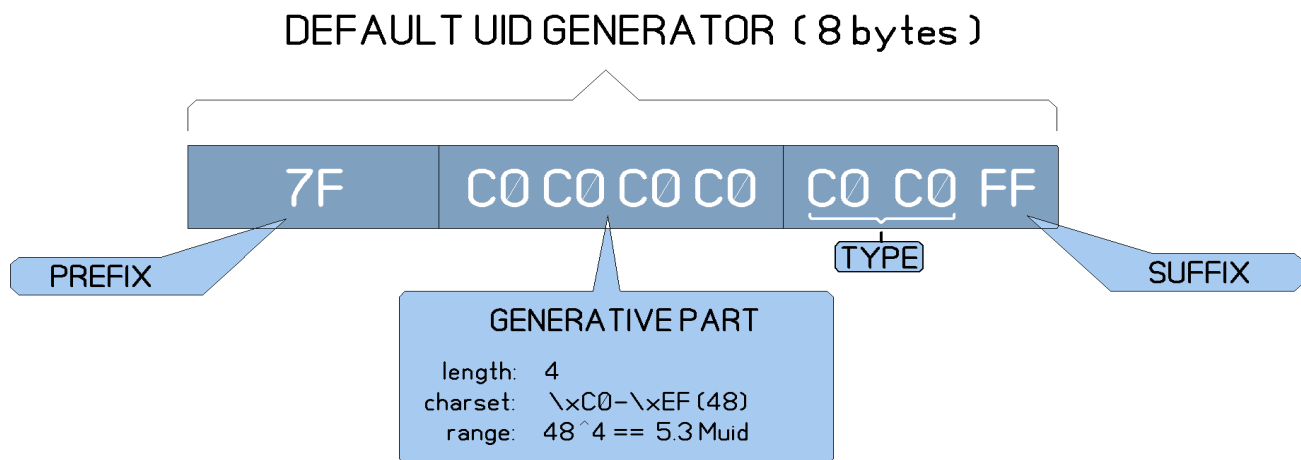
Оба предустановленных uid-генератора нельзя отменить или переобъявить..

Все другие uid-генераторы могут быть свободно переобъявляемы в любом режиме.

## DEFAULT UID-GENERATOR

указатель: `""` (пустая строка)

Дефолтовый uid-генератор генерирует строки со следующей структурой:



**Diagram 2**

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

Дефолтовый uid-генератор необходим библиотеке для нормального функционирования и не может быть отменён или переобъявлен.

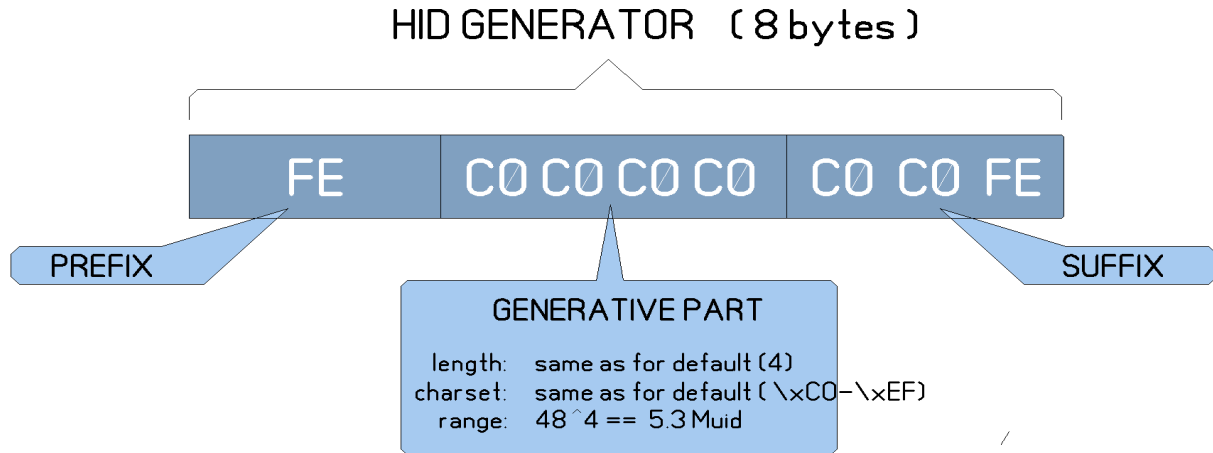
Смотрите раздел **POINTERS IN AWK**.



## HID-GENERATOR

указатель: "hid"

Hid-генератор генерирует строки со следующей структурой:



**Diagram 3**

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

## HIDs IN AWK

**Hid** является **очень важным ресурсом** программирования в AWK.

Всякий раз когда вы собираетесь **сохранить** в некотором **массиве** какие-то сторонние **данные** — вы вынуждены раз за разом придумывать ответ на один и тот-же вопрос: **под какими индексами** это делать?

Действительно, ведь в массиве находятся некоторые **«рабочие» индексы** — нам крайне необходимо чтобы «служебные» индексы никак **не конфликтовали** с возможными «рабочими» индексами.

И если мы просто, допустим, будем указывать «служебные» индексы **прямой строкой**:

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

... то по понятным причинам мы очень быстро на такой **конфликт** как раз и нарвёмся.

Важным также остаётся и **читабельность кода**: если вы начнёте усложнять строку индекса в массиве какими-то дополнительными символами - чтобы минимизировать вероятность **возможных конфликтов** с рабочими индексами — то вы неизбежно потеряете её в читабельности:

```
A[ "\xFEELIST\xFE" ]
```

Возникает **замкнутый круг**: мы хотим предельно **ясной записи индекса**, но такая запись может вступить в **конфликт с рабочими индексами**. Если-же мы будем усложнять индекс, то неизбежно будем терять в комфорте.

**HID** является системным решением данной проблемы.

Hid — это глобальная переменная - называемая «несущей» - которая **объявляется с помощью функции hid()** и которой присваивается так называемое **сильное значение** генерируемое **hid-генератором**.

Суть присваиваемого hid-переменной значения заключается в том, **чтобы не быть похожим ни на что** — ни на возможные данные, ни на возможные «рабочие» индексы в массивах.

Пользователь может активно использовать несущие hid-переменные как своего рода **«служебные» индексы** в массивах - для доступа к какой-либо дополнительной, сопутствующей информации сохраняемой **внутри массива**:

```
A[ SOME_ ]      вместо:      A[ "SOME_" ]
```

**ВАЖНО!** Негативным фактором при использовании hid является тот факт, что если **hid** переменная не объявлена или её имя написано с ошибкой - то пользователь об этом **никак прямо не узнает** и код начнёт выполнять **непредсказуемые действия** (необъявленная переменная будет возвращать пустую строку вместо специального значения).

Поэтому следует **внимательно относиться к написанию hid** в коде и следить за тем чтобы **все используемые hid были предварительно объявлены**.

Некоторые служебные индексы которые нам понадобятся: - локальные: т. е. **внутренние в указанном namespace**. А некоторые, нужны такие, чтобы в любом namespace объявлялись **одним и тем-же значение(глобальные)**.

Объявления hid должны производиться **в полном объёме в каждом namespace** где они используются.

Так называемые **суперглобальные hid** – несущими переменными которых являются суперглобальные переменные — также **обязательны для объявления в каждом namespace** в которых они используются.

**Объявление hid** производится с помощью присваивания несущей hid-переменной результата возвращаемого функцией **hid()** вызванной со строкой имени объявляемого hid в качестве параметра:

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

example:

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

```
print hexdump( SOME_ )
```

outputs (например):

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

Объявляется **hid** с именем: *name*. Несущей hid-переменной *name* присваивается некоторое специальное значение.

Параметр *code* позволяет пользователю произвести **объявление hid** используя собственное **специальное значение**. Если *code* != «», то оно становится **новым специальным значением**. В противном случае специальное **новое значение** будет сгенерировано **hid-генератором**.

В некоторых случаях пользователю просто нужно **специальное значение**, но не нужна его регистрация как hid:

> Если параметр *name* равен **пустой строке** ( == "" ), то функция **hid()** просто **возвращает новое специальное значение**:

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

Параметр *name* содержит имя объявляемого hid. Имя может быть дано с указанием namespace или без него. В качестве **namespace-разделителя** можно использовать ':' или '!'.

Если *hid name* до этого момента объявлен не был, то функция `hid()` **вернёт новое специальное значение и сохранит** его в ассоциации с *name* .

При дальнейших попытках объявления того-же *name* функция `hid::get()` будет просто **возвращать сохранённое значение**.

> Если *name* не содержит **namespace-разделителя**, то производится **объявление глобального hid**:

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

**Глобальный hid в любом namespace несёт одно и то-же значение.**

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

> Если параметр *name* содержит **namespace-разделитель**, то производится **объявление локального hid**:

```
@namespace "some"
BEGIN{
    LOCAL_ = hid::get( "some::LOCAL_" )
}
```

**Локальный hid объявляется только для своего namespace и всегда несёт своё собственное значение.**

> Если в параметре *name* **не указывается namespace**, то используется тот namespace, который **указывался функции `hid::get()` последним**:

```
@namespace "some"
BEGIN{
    LOCAL1_ = hid::get( "some.LOCAL_" )
    LOCAL2_ = hid::get( ".LOCAL2_" )
}
```

## HID[]

Для того чтобы понять является-ли некоторая строка одним из «служебных» индексов пользователь может применять проверку экзистенса этой строки в качестве индекса в суперглобальном массиве **HID**.

Например внутри for-in цикла:

```
for ( i in A )  
  
    if ( i in HID )  
  
        continue  
  
    else ...
```

Суперглобальный массив **HID** содержит в качестве индексов все специальные значения присвоенные hid.

### hid::name( code )

Параметр *code* должен быть значением одной из объявленных несущих hid-переменных.

Если *code* не является таким значением, то функция возвращает **null**.

Функция возвращает имя hid с кодом *code*.

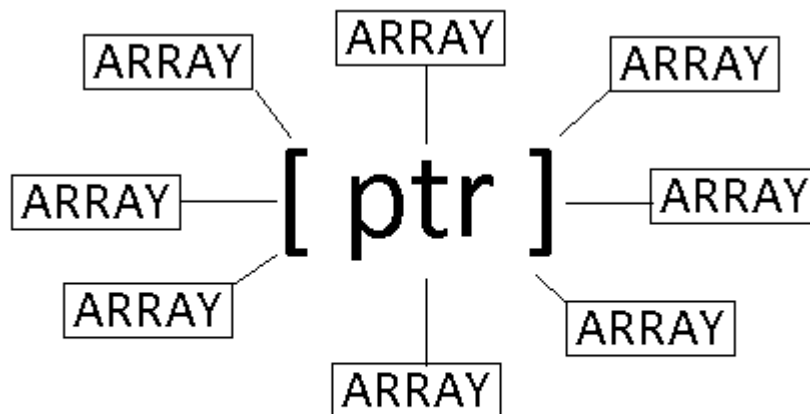
example:

```
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



### object in AWK

Для лучшего понимания того, как использовать указатели в AWK рекомендуется прежде всего ознакомиться с исходным кодом самой библиотеки:

- вы увидите как следует располагать данные относительно указателя:

```
FIELD[ ptr ]
```

Например: некоторый указатель *ptr* имеет обязательное свойство: ID  
располагаемый в:

```
ID[ ptr ]
```

— как узнать: является-ли *ptr* указателем:

Для определения является-ли строка *ptr* действующим указателем необходимо выполнить:

```
if ( ptr in ID )  
    # ptr указатель  
else # ptr что-то другое
```

Тела указателей имеют характерную структуру легко распознаваемую с помощью регулярных выражений среди данных: каждый указатель начинается с символа `\x7F` и затем через 6 байт, заканчивается символом `\xFF`.

Таким образом «что-то другое» - в примере выше - может означать что строка *ptr* это любая другая строка состоящая из данных и указателей.

Суть предлагаемой методики работы с данными состоит в том, что исходная строка для нас: это смесь данных и указателей.

Для поиска указателей любого типа предлагается использовать регулярное выражение:

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

Если искать надо указатели конкретного типа(ов), то следует указывать возможные символы в поле типа искомого указателя (два предпоследних байта в указателе — см. *Diagram 2*):

```
/\x7F....\xT0\xT1\xff/
```

где: \xT0 и \xT1 — символы-идентификаторы типа (класса) искомого указателя

Для того, чтобы «не наехать» каким-либо сторонним регулярным выражением на тела указателей лежащих среди данных следует избегать символа \x7F (при движении вправо) и символа \xFF (при движении влево).

## FYI

Uid-генераторы в моём энвайрименте также используются для генерации имён временных файлов

Настоятельно не советую экспериментировать с настройками дефолтового и hid генераторов: поверьте, взятые на вооружение структуры генерируемых ими строк — взяты не с потолка или откуда-нибудь. Для меня лично они являются указателями уже третьего поколения.

## ПЕРСПЕКТИВЫ

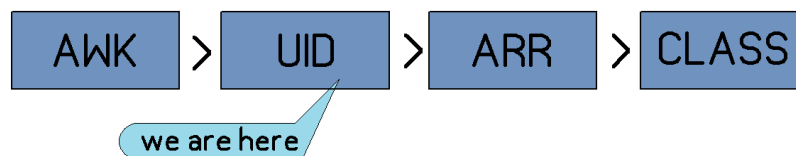
На первом этапе нам необходимо поднять общий уровень базовой техники программирования в AWK.

Данная библиотека относится к базовой технике программирования на AWK

Следующим релизом станет библиотека ARR — для работы с массивами (также базовая техника).

Затем последует релиз библиотеки CLASS — представляющая из себя классовую имплементацию для AWK: основы базовой техники и описание приёмов ООП в AWK.

Гипотетический план публикаций библиотек для AWK на 2024-2025 год:



## **AUTHOR**

Классовая имплантация в AWK является ключевым моментом для дальнейшего всестороннего развития этого языка.

Моя личная «галактика Андромеды» находится в области работы с комплексным регулярным выражением, и именно туда я вас веду.

Благодарю Вас за интерес и за Ваше внимание!

That's all folks! :)

*Kind Regards*  
Denis Shirokov

## **RESPECT DUE**

All awkers from all the world!

gawk Team