

# **uLan Utilities Library (ULUT)**

**Pavel Pisa**

**`pisa@cmp.felk.cvut.cz`**

## **uLan Utilities Library (ULUT)**

by Pavel Pisa

Copyright © 2003-2006 Pavel Pisa

The ULUT library contains useful constructions and functions commonly used in "C" language programs. It tries to be something like STL library is for the C++ programs, but is is fall less simpler and powerful. It has small memory footprint and could be used even for small embedded applications. Basic functionalities provided by library are more types of containers (list, AVL tree, priority queue, heap-tree, etc.), timer framework and event connectors.

# Table of Contents

<b>1. Functions Description.....</b>	<b>1</b>
1.1. Generic AVL tree .....	1
struct gavl_node.....	1
struct gavl_root.....	2
gavl_node2item .....	2
gavl_node2item_safe .....	3
gavl_node2key.....	4
gavl_next_node.....	5
gavl_prev_node.....	5
gavl_balance_one .....	6
gavl_insert_primitive_at.....	6
gavl_delete_primitive .....	7
gavl_cut_first_primitive.....	8
gavl_first_node .....	9
gavl_last_node.....	10
gavl_is_empty.....	10
gavl_search_node .....	11
gavl_find .....	12
gavl_find_first.....	13
gavl_find_after.....	13
gavl_insert_node_at.....	14
gavl_insert_node.....	15
gavl_insert .....	16
gavl_delete_node .....	17
gavl_delete.....	17
gavl_delete_and_next_node .....	18
gavl_cut_first .....	19
1.2. Hierarchical Timer Framework .....	20
struct ul_htim_node .....	20
struct ul_htim_queue .....	21
struct ul_htimer.....	21
struct ul_htimer_queue .....	22
1.3. Generic Sorted Arrays.....	23
struct gsa_array_field_t .....	23
gsa_struct_init.....	24
gsa_delete_all .....	24
gsa_bsearch_indx .....	25
gsa_find.....	26
gsa_find_first .....	27
gsa_find_indx .....	28
gsa_insert_at.....	28
gsa_insert.....	29
gsa_delete_at .....	30
gsa_delete .....	31
gsa_resort_buble.....	31
gsa_setsort .....	32

1.4. Bidirectional Linked List .....	33
list_add .....	33
list_add_tail .....	34
list_del .....	34
list_del_init .....	35
list_move .....	36
list_move_tail .....	36
list_empty .....	37
list_splice .....	37
list_splice_init .....	38
list_entry .....	39
list_for_each .....	39
__list_for_each .....	40
list_for_each_prev .....	40
list_for_each_safe .....	41
list_for_each_entry .....	42
list_for_each_entry_reverse .....	42
list_for_each_entry_safe .....	43
1.5. Dynamic Buffers .....	44
struct ul_dbuff .....	44
ul_dbuff_init .....	45
ul_dbuff_destroy .....	45
ul_dbuff_prep .....	46
ul_dbuff_set_capacity .....	46
ul_dbuff_set_len .....	47
ul_dbuff_set .....	48
ul_dbuff_cpy .....	49
ul_dbuff_cat .....	49
ul_dbuff_strcat .....	50
ul_dbuff_strcpy .....	51
ul_dbuff_append_byte .....	52
ul_dbuff_ltrim .....	52
ul_dbuff_rtrim .....	53
ul_dbuff_trim .....	54
ul_dbuff_cpos .....	54
ul_str_cpos .....	55
ul_str_pos .....	56
ul_str_ncpy .....	57
ul_dbuff_cut_pos .....	58
ul_dbuff_cut_delimited .....	58
ul_dbuff_cut_token .....	59
ul_dbuff_export .....	60
1.6. Event Connectors .....	61
struct evc_link .....	61
struct evc_tx_hub .....	63
struct evc_rx_hub .....	63
evc_link_inc_refcnt .....	64
evc_link_dec_refcnt .....	64

evc_link_init .....	65
evc_link_new .....	66
evc_link_connect .....	67
evc_link_init_standalone .....	68
evc_link_new_standalone .....	69
evc_link_connect_standalone .....	69
evc_link_delete .....	70
evc_link_dispose .....	71
evc_tx_hub_init .....	72
evc_tx_hub_done .....	72
evc_tx_hub_propagate .....	73
evc_tx_hub_emit .....	73
evc_rx_hub_init .....	74
evc_rx_hub_disconnect_all .....	75
evc_rx_hub_done .....	75
1.7. Application Messages Logging .....	76
struct ul_log_domain .....	76
ul_log1 .....	77
ul_log_redir .....	78
1.8. Unique IDs Generator .....	78
struct ul_uniqid_pool_t .....	79
ul_uniqid_pool_init .....	79
ul_uniqid_pool_done .....	80
ul_uniqid_pool_reserve .....	81
ul_uniqid_pool_release .....	82
ul_uniqid_pool_alloc_one .....	83
ul_uniqid_pool_alloc_one_after .....	84
ul_uniqid_pool_free_one .....	85

# Chapter 1. Functions Description

## 1.1. Generic AVL tree

### struct gavl\_node

#### Name

struct gavl\_node — Structure Representing Node of Generic AVL Tree

#### Synopsis

```
struct gavl_node {
    struct gavl_node * left;
    struct gavl_node * right;
    struct gavl_node * parent;
    int hdiff;
};
```

#### Members

left

pointer to left child or NULL

right

pointer to right child or NULL

parent

pointer to parent node, NULL for root

hdiff

difference of height between left and right child

#### Description

This structure represents one node in the tree and links *left* and *right* to nodes with lower and higher value of order criterion. Each tree is built from one type of items defined by user. User can decide to include node structure inside item representation or GAVL can malloc node structures for each inserted

item. The GAVL allocates memory space with capacity `sizeof(gavl_node_t)+sizeof(void*)` in the second case. The item pointer is stored following node structure `(void**)(node+1)`;

## struct gavl\_root

### Name

`struct gavl_root` — Structure Representing Root of Generic AVL Tree

### Synopsis

```
struct gavl_root {
    gavl_node_t * root_node;
    int node_offs;
    int key_offs;
    gavl_cmp_fnc_t * cmp_fnc;
};
```

### Members

`root_node`

pointer to root node of GAVL tree

`node_offs`

offset between start of user defined item representation and included GAVL node structure. If negative value is stored there, user item does not contain node structure and GAVL manages standalone ones with item pointers.

`key_offs`

offset to compared (ordered) fields in the item representation

`cmp_fnc`

function defining order of items by comparing fields at offset `key_offs`.

## **gavl\_node2item**

### **Name**

`gavl_node2item` — Conversion from GAVL Tree Node to User Defined Item

### **Synopsis**

```
void * gavl_node2item (const gavl_root_t * root, const gavl_node_t * node);
```

### **Arguments**

*root*

GAVL tree root

*node*

node belonging to *root* GAVL tree

### **Return Value**

pointer to item corresponding to node

## **gavl\_node2item\_safe**

### **Name**

`gavl_node2item_safe` — Conversion from GAVL Tree Node to User Defined Item

### **Synopsis**

```
void * gavl_node2item_safe (const gavl_root_t * root, const gavl_node_t *  
node);
```



## Arguments

*root*

GAVL tree root

*node*

node belonging to *root* GAVL tree

## Return Value

pointer to item corresponding to node

# **gavl\_node2key**

## Name

`gavl_node2key` — Conversion from GAVL Tree Node to Ordering Key

## Synopsis

```
void * gavl_node2key (const gavl_root_t * root, const gavl_node_t * node);
```

## Arguments

*root*

GAVL tree root

*node*

node belonging to *root* GAVL tree

## Return Value

pointer to key corresponding to node

## **gavl\_next\_node**

### **Name**

`gavl_next_node` — Returns Next Node of GAVL Tree

### **Synopsis**

```
gavl_node_t * gavl_next_node (const gavl_node_t * node);
```

### **Arguments**

*node*

node for which accessor is looked for

### **Return Value**

pointer to next node of tree according to ordering

## **gavl\_prev\_node**

### **Name**

`gavl_prev_node` — Returns Previous Node of GAVL Tree

### **Synopsis**

```
gavl_node_t * gavl_prev_node (const gavl_node_t * node);
```

## Arguments

*node*

node for which predecessor is looked for

## Return Value

pointer to previous node of tree according to ordering

# gavl\_balance\_one

## Name

`gavl_balance_one` — Balance One Node to Enhance Balance Factor

## Synopsis

```
int gavl_balance_one (gavl_node_t ** subtree);
```

## Arguments

*subtree*

pointer to pointer to node for which balance is enhanced

## Return Value

returns nonzero value if height of subtree is lowered by one

# **gavl\_insert\_primitive\_at**

## **Name**

`gavl_insert_primitive_at` — Low Level Routine to Insert Node into Tree

## **Synopsis**

```
int gavl_insert_primitive_at (gavl_node_t ** root_nodep, gavl_node_t * node,  
gavl_node_t * where, int leftright);
```

## **Arguments**

*root\_nodep*

pointer to pointer to GAVL tree root node

*node*

pointer to inserted node

*where*

pointer to found parent node

*leftright*

left ( $\geq 1$ ) or right ( $\leq 0$ ) branch

## **Description**

This function can be used for implementing AVL trees with custom root definition. The value of the selected *left* or *right* pointer of provided *node* has to be NULL before insert operation, i.e. node has to be end node in the selected direction.

## **Return Value**

positive value informs about success

## **gavl\_delete\_primitive**

### **Name**

`gavl_delete_primitive` — Low Level Deletes/Unlinks Node from GAVL Tree

### **Synopsis**

```
int gavl_delete_primitive (gavl_node_t ** root_nodep, gavl_node_t * node);
```

### **Arguments**

*root\_nodep*

pointer to pointer to GAVL tree root node

*node*

pointer to deleted node

### **Return Value**

positive value informs about success.

## **gavl\_cut\_first\_primitive**

### **Name**

`gavl_cut_first_primitive` — Low Level Routine to Cut First Node from Tree

### **Synopsis**

```
gavl_node_t * gavl_cut_first_primitive (gavl_node_t ** root_nodep);
```

## Arguments

*root\_nodep*

pointer to pointer to GAVL tree root node

## Description

This enables fast delete of the first node without tree balancing. The resulting tree is degraded but height differences are kept consistent. Use of this function can result in height of tree maximally one greater the tree managed by optimal AVL functions.

## Return Value

returns the first node or NULL if the tree is empty

# **gavl\_first\_node**

## Name

`gavl_first_node` — Returns First Node of GAVL Tree

## Synopsis

```
gavl_node_t * gavl_first_node (const gavl_root_t * root);
```

## Arguments

*root*

GAVL tree root

## Return Value

pointer to the first node of tree according to ordering

## **gavl\_last\_node**

### **Name**

`gavl_last_node` — Returns Last Node of GAVL Tree

### **Synopsis**

```
gavl_node_t * gavl_last_node (const gavl_root_t * root);
```

### **Arguments**

*root*

GAVL tree root

### **Return Value**

pointer to the last node of tree according to ordering

## **gavl\_is\_empty**

### **Name**

`gavl_is_empty` — Check for Empty GAVL Tree

### **Synopsis**

```
int gavl_is_empty (const gavl_root_t * root);
```

## Arguments

*root*

GAVL tree root

## Return Value

returns non-zero value if there is no node in the tree

# **gavl\_search\_node**

## Name

`gavl_search_node` — Search for Node or Place for Node by Key

## Synopsis

```
int gavl_search_node (const gavl_root_t * root, const void * key, int mode,  
gavl_node_t ** nodep);
```

## Arguments

*root*

GAVL tree root

*key*

key value searched for

*mode*

mode of the search operation

*nodep*

pointer to place for storing of pointer to found node or pointer to node which should be parent of inserted node



## Description

Core search routine for GAVL trees searches in tree starting at *root* for node of item with value of item field at offset *key\_off* equal to provided *\*key* value. Values are compared by function pointed by *\*cmp\_fnc* field in the tree *root*. Integer *mode* modifies search algorithm: *GAVL\_FANY* .. finds node of any item with field value *\*key*, *GAVL\_FFIRST* .. finds node of first item with *\*key*, *GAVL\_FAFTER* .. node points after last item with *\*key* value, reworded - index points at first item with higher value of field or after last item

## Return Value

Return of nonzero value indicates match found. If the *mode* is ored with *GAVL\_FCMP*, result of last compare is returned.

## `gavl_find`

### Name

`gavl_find` — Find Item for Provided Key

### Synopsis

```
void * gavl_find (const gavl_root_t * root, const void * key);
```

### Arguments

*root*

GAVL tree root

*key*

key value searched for

### Return Value

pointer to item associated to key value.

## **gavl\_find\_first**

### **Name**

`gavl_find_first` — Find the First Item with Provided Key Value

### **Synopsis**

```
void * gavl_find_first (const gavl_root_t * root, const void * key);
```

### **Arguments**

*root*

GAVL tree root

*key*

key value searched for

### **Description**

same as above, but first matching item is found.

### **Return Value**

pointer to the first item associated to key value.

## **gavl\_find\_after**

### **Name**

`gavl_find_after` — Find the First Item with Higher Key Value

## Synopsis

```
void * gavl_find_after (const gavl_root_t * root, const void * key);
```

## Arguments

*root*

GAVL tree root

*key*

key value searched for

## Description

same as above, but points to item with first key value above searched *key*.

## Return Value

pointer to the first item associated to key value.

## **gavl\_insert\_node\_at**

### Name

`gavl_insert_node_at` — Insert Existing Node to Already Computed Place into GAVL Tree

### Synopsis

```
int gavl_insert_node_at (gavl_root_t * root, gavl_node_t * node, gavl_node_t  
* where, int leftright);
```

## Arguments

*root*

GAVL tree root

*node*

pointer to inserted node

*where*

pointer to found parent node

*leftright*

left (1) or right (0) branch

## Return Value

positive value informs about success

# **gavl\_insert\_node**

## Name

`gavl_insert_node` — Insert Existing Node into GAVL Tree

## Synopsis

```
int gavl_insert_node (gavl_root_t * root, gavl_node_t * node, int mode);
```

## Arguments

*root*

GAVL tree root

*node*

pointer to inserted node

*mode*

if mode is `GAVL_FASTER`, multiple items with same key can be used, else strict ordering is required

## Return Value

positive value informs about success

# **gavl\_insert**

## Name

`gavl_insert` — Insert New Item into GAVL Tree

## Synopsis

```
int gavl_insert (gavl_root_t * root, void * item, int mode);
```

## Arguments

*root*

GAVL tree root

*item*

pointer to inserted item

*mode*

if mode is `GAVL_FASTER`, multiple items with same key can be used, else strict ordering is required

## Return Value

positive value informs about success, negative value indicates malloc fail or attempt to insert item with already defined key.

## **gavl\_delete\_node**

### **Name**

`gavl_delete_node` — Deletes/Unlinks Node from GAVL Tree

### **Synopsis**

```
int gavl_delete_node (gavl_root_t * root, gavl_node_t * node);
```

### **Arguments**

*root*

GAVL tree root

*node*

pointer to deleted node

### **Return Value**

positive value informs about success.

## **gavl\_delete**

### **Name**

`gavl_delete` — Delete/Unlink Item from GAVL Tree

### **Synopsis**

```
int gavl_delete (gavl_root_t * root, void * item);
```

## Arguments

*root*

GAVL tree root

*item*

pointer to deleted item

## Return Value

positive value informs about success, negative value indicates that item is not found in tree defined by *root*

# **`gavl_delete_and_next_node`**

## Name

`gavl_delete_and_next_node` — Delete/Unlink Item from GAVL Tree

## Synopsis

```
gavl_node_t * gavl_delete_and_next_node (gavl_root_t * root, gavl_node_t *  
node);
```

## Arguments

*root*

GAVL tree root

*node*

pointer to actual node which is unlinked from tree after function call, it can be unallocated or reused by application code after this call.

## Description

This function can be used after call `gavl_first_node` for destructive traversal through the tree, it cannot be combined with `gavl_next_node` or `gavl_prev_node` and root is emptied after the end of traversal. If the tree is used after unsuccessful/unfinished traversal, it must be balanced again. The height differences are inconsistent in other case. If traversal could be interrupted, the function `gavl_cut_first` could be better choice.

## Return Value

pointer to next node or NULL, when all nodes are deleted

## `gavl_cut_first`

### Name

`gavl_cut_first` — Cut First Item from Tree

### Synopsis

```
void * gavl_cut_first (gavl_root_t * root);
```

### Arguments

*root*

GAVL tree root

### Description

This enables fast delete of the first item without tree balancing. The resulting tree is degraded but height differences are kept consistent. Use of this function can result in height of tree maximally one greater the tree managed by optimal AVL functions.



## Return Value

returns the first item or NULL if the tree is empty

## 1.2. Hierarchical Timer Framework

### struct ul\_htim\_node

#### Name

struct ul\_htim\_node — Timer queue entry base structure

#### Synopsis

```
struct ul_htim_node {
#ifdef UL_HTIMER_WITH_HPTREE
    ul_hpt_node_t node;
#else
    ul_hpt_node_t node;
#endif
    ul_htim_time_t expires;
};
```

#### Members

node

regular GAVL node structure for insertion into

node

regular GAVL node structure for insertion into

expires

time to trigger timer in &ul\_htim\_time\_t type defined resolution

#### Description

This is basic type useful to define more complete timer types

## struct ul\_htim\_queue

### Name

struct ul\_htim\_queue — Timer queue head/root base structure

### Synopsis

```
struct ul_htim_queue {
#ifdef UL_HTIMER_WITH_HPTREE
    ul_hpt_root_field_t timers;
#else
    ul_hpt_root_field_t timers;
#endif
    int first_changed;
};
```

### Members

timers

root of FLES GAVL tree of timer entries

timers

root of FLES GAVL tree of timer entries

first\_changed

flag, which is set after each add, detach operation which concerning of firsts scheduled timer

### Description

This is basic type useful to define more complete timer queues types

## struct ul\_htimer

### Name

`struct ul_htimer` — Standard timer entry with callback function

### Synopsis

```
struct ul_htimer {
    ul_hitim_node_t htim;
    ul_htimer_fnc_t * function;
    unsigned long data;
};
```

### Members

`htim`

basic timer queue entry

`function`

user provided function to call at trigger time

`data`

user selected data

### Description

This is standard timer type, which requires *data* casting in many cases. The type of *function* field has to be declared in “ul\_hitimdefs.h” header file.

## struct ul\_htimer\_queue

### Name

`struct ul_htimer_queue` — Standard timer queue

## Synopsis

```
struct ul_htimer_queue {
    ul_htim_queue_t htim_queue;
};
```

## Members

htim\_queue

the structure wraps &ul\_htim\_queue structure

## Description

This is standard timer type, which requires *data* casting in many cases

# 1.3. Generic Sorted Arrays

## struct gsa\_array\_field\_t

### Name

struct gsa\_array\_field\_t — Structure Representing Anchor of custom GSA Array

### Synopsis

```
struct gsa_array_field_t {
    void ** items;
    unsigned count;
    unsigned alloc_count;
};
```

### Members

items

pointer to array of pointers to individual items

`count`  
 number of items in the sorted array

`alloc_count`  
 allocated pointer array capacity

## **gsa\_struct\_init**

### **Name**

`gsa_struct_init` — Initialize GSA Structure

### **Synopsis**

```
void gsa_struct_init (gsa_array_t * array, int key_offs, gsa_cmp_fnc_t *
cmp_fnc);
```

### **Arguments**

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*key\_offs*

offset to the order controlling field obtained by `UL_OFFSETOF`

*cmp\_fnc*

function defining order of items by comparing fields at offset *key\_offs*.

## **gsa\_delete\_all**

### **Name**

`gsa_delete_all` — Delete Pointers to the All Items in the Array

## Synopsis

```
void gsa_delete_all (gsa_array_t * array);
```

## Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

## Description

This function releases all internally allocated memory, but does not release memory of the *array* structure

# gsa\_bsearch\_indx

## Name

`gsa_bsearch_indx` — Search for Item or Place for Item by Key

## Synopsis

```
int gsa_bsearch_indx (gsa_array_t * array, void * key, int key_offs,  
gsa_cmp_fnc_t * cmp_fnc, int mode, unsigned * indx);
```

## Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*key*

key value searched for

*key\_offs*offset to the order controlling field obtained by `UL_OFFSETOF`*cmp\_fnc*

function defining order of items by comparing fields

*mode*

mode of the search operation

*indx*

pointer to place, where store value of found item array index or index where new item should be inserted

## Description

Core search routine for GSA arrays binary searches for item with field at offset *key\_off* equal to *key* value. Values are compared by function pointed by *\*cmp\_fnc* field in the array structure *array*. Integer *mode* modifies search algorithm: `GSA_FANY` .. finds item with field value *\*key*, `GSA_FFIRST` .. finds the first item with field value *\*key*, `GSA_FAFTER` .. index points after last item with *\*key* value, reworded - index points at first item with higher value of field or after last item

## Return Value

Return of nonzero value indicates match found.

## gsa\_find

### Name

`gsa_find` — Find Item for Provided Key

### Synopsis

```
void * gsa_find (gsa_array_t * array, void * key);
```

## Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*key*

key value searched for

## Return Value

pointer to item associated to key value or `NULL`.

## **gsa\_find\_first**

### Name

`gsa_find_first` — Find the First Item for Provided Key

### Synopsis

```
void * gsa_find_first (gsa_array_t * array, void * key);
```

## Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*key*

key value searched for

## Description

same as above, but first matching item is found.



## Return Value

pointer to the first item associated to key value or NULL.

## gsa\_find\_indx

### Name

`gsa_find_indx` — Find the First Item with Key Value and Return Its Index

### Synopsis

```
void * gsa_find_indx (gsa_array_t * array, void * key, int * indx);
```

### Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*key*

key value searched for

*indx*

pointer to place for index, at which new item should be inserted

### Description

same as above, but additionally stores item index value.

### Return Value

pointer to the first item associated to key value or NULL.

## gsa\_insert\_at

### Name

`gsa_insert_at` — Insert Existing Item to the Specified Array Index

### Synopsis

```
int gsa_insert_at (gsa_array_t * array, void * item, unsigned where);
```

### Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*item*

pointer to inserted Item

*where*

at which index should be *item* inserted

### Return Value

positive or zero value informs about success

## gsa\_insert

### Name

`gsa_insert` — Insert Existing into Ordered Item Array

### Synopsis

```
int gsa_insert (gsa_array_t * array, void * item, int mode);
```

## Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*item*

pointer to inserted Item

*mode*

if mode is `GSA_FASTER`, multiple items with same key can be stored into array, else strict ordering is required

## Return Value

positive or zero value informs about success

## `gsa_delete_at`

### Name

`gsa_delete_at` — Delete Item from the Specified Array Index

### Synopsis

```
int gsa_delete_at (gsa_array_t * array, unsigned indx);
```

### Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*indx*

index of deleted item

## Return Value

positive or zero value informs about success

## gsa\_delete

### Name

`gsa_delete` — Delete Item from the Array

### Synopsis

```
int gsa_delete (gsa_array_t * array, void * item);
```

### Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*item*

pointer to deleted Item

### Return Value

positive or zero value informs about success

## gsa\_resort\_buble

### Name

`gsa_resort_buble` — Sort Again Array If Sorting Criteria Are Changed

### Synopsis

```
int gsa_resort_buble (gsa_array_t * array, int key_offs, gsa_cmp_fnc_t *
cmp_fnc);
```

### Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*key\_offs*

offset to the order controlling field obtained by `UL_OFFSETOF`

*cmp\_fnc*

function defining order of items by comparing fields

### Return Value

non-zero value informs, that resorting changed order

## gsa\_setsort

### Name

`gsa_setsort` — Change Array Sorting Criterion

### Synopsis

```
int gsa_setsort (gsa_array_t * array, int key_offs, gsa_cmp_fnc_t * cmp_fnc);
```

## Arguments

*array*

pointer to the array structure declared through `GSA_ARRAY_FOR`

*key\_offs*

new value of offset to the order controlling field

*cmp\_fnc*

new function defining order of items by comparing fields at offset *key\_offs*

## Return Value

non-zero value informs, that resorting changed order

## 1.4. Bidirectional Linked List

### list\_add

#### Name

`list_add` — add a new entry

#### Synopsis

```
void list_add (struct list_head * newe, struct list_head * head);
```

#### Arguments

*newe*

-- undescribed --

*head*

list head to add it after

## Description

Insert a new entry after the specified head. This is good for implementing stacks.

## list\_add\_tail

### Name

`list_add_tail` — add a new entry

### Synopsis

```
void list_add_tail (struct list_head * newe, struct list_head * head);
```

### Arguments

*newe*

new entry to be added

*head*

list head to add it before

### Description

Insert a new entry before the specified head. This is useful for implementing queues.

## list\_del

### Name

`list_del` — deletes entry from list.

### Synopsis

```
void list_del (struct list_head * entry);
```

### Arguments

*entry*

the element to delete from the list.

### Note

`list_empty` on `entry` does not return true after this, the entry is in an undefined state.

## list\_del\_init

### Name

`list_del_init` — deletes entry from list and reinitialize it.

### Synopsis

```
void list_del_init (struct list_head * entry);
```



## Arguments

*entry*

the element to delete from the list.

## list\_move

### Name

`list_move` — delete from one list and add as another's head

### Synopsis

```
void list_move (struct list_head * list, struct list_head * head);
```

## Arguments

*list*

the entry to move

*head*

the head that will precede our entry

## list\_move\_tail

### Name

`list_move_tail` — delete from one list and add as another's tail

### Synopsis

```
void list_move_tail (struct list_head * list, struct list_head * head);
```

## Arguments

*list*

the entry to move

*head*

the head that will follow our entry

## list\_empty

### Name

`list_empty` — tests whether a list is empty

### Synopsis

```
int list_empty (struct list_head * head);
```

### Arguments

*head*

the list to test.

## list\_splice

### Name

`list_splice` — join two lists

## Synopsis

```
void list_splice (struct list_head * list, struct list_head * head);
```

## Arguments

*list*

the new list to add.

*head*

the place to add it in the first list.

## list\_splice\_init

### Name

`list_splice_init` — join two lists and reinitialise the emptied list.

## Synopsis

```
void list_splice_init (struct list_head * list, struct list_head * head);
```

## Arguments

*list*

the new list to add.

*head*

the place to add it in the first list.

## Description

The list at *list* is reinitialised

## list\_entry

### Name

`list_entry` — get the struct for this entry

### Synopsis

```
list_entry ( ptr, type, member);
```

### Arguments

*ptr*

the &struct list\_head pointer.

*type*

the type of the struct this is embedded in.

*member*

the name of the list\_struct within the struct.

## list\_for\_each

### Name

`list_for_each` — iterate over a list

### Synopsis

```
list_for_each ( pos, head);
```

## Arguments

*pos*

the `&struct list_head` to use as a loop counter.

*head*

the head for your list.

## `__list_for_each`

### Name

`__list_for_each` — iterate over a list

### Synopsis

```
__list_for_each ( pos, head );
```

## Arguments

*pos*

the `&struct list_head` to use as a loop counter.

*head*

the head for your list.

## Description

This variant differs from `list_for_each` in that it's the simplest possible list iteration code, no prefetching is done. Use this for code that knows the list to be very short (empty or 1 entry) most of the time.

## list\_for\_each\_prev

### Name

`list_for_each_prev` — iterate over a list backwards

### Synopsis

```
list_for_each_prev ( pos, head );
```

### Arguments

*pos*

the &struct list\_head to use as a loop counter.

*head*

the head for your list.

## list\_for\_each\_safe

### Name

`list_for_each_safe` — iterate over a list safe against removal of list entry

### Synopsis

```
list_for_each_safe ( pos, n, head );
```

### Arguments

*pos*

the &struct list\_head to use as a loop counter.

*n*  
another &struct list\_head to use as temporary storage

*head*  
the head for your list.

## list\_for\_each\_entry

### Name

`list_for_each_entry` — iterate over list of given type

### Synopsis

```
list_for_each_entry ( pos, head, member );
```

### Arguments

*pos*  
the type \* to use as a loop counter.

*head*  
the head for your list.

*member*  
the name of the list\_struct within the struct.

## list\_for\_each\_entry\_reverse

### Name

`list_for_each_entry_reverse` — iterate backwards over list of given type.

## Synopsis

```
list_for_each_entry_reverse ( pos, head, member );
```

## Arguments

*pos*

the type \* to use as a loop counter.

*head*

the head for your list.

*member*

the name of the list\_struct within the struct.

## list\_for\_each\_entry\_safe

### Name

`list_for_each_entry_safe` — iterate over list of given type safe against removal of list entry

## Synopsis

```
list_for_each_entry_safe ( pos, n, head, member );
```

## Arguments

*pos*

the type \* to use as a loop counter.

*n*

another type \* to use as temporary storage

*head*

the head for your list.



*member*

the name of the list\_struct within the struct.

## 1.5. Dynamic Buffers

### struct ul\_dbuff

#### Name

struct ul\_dbuff — Generic Buffer for Dynamic Data

#### Synopsis

```
struct ul_dbuff {
    unsigned long len;
    unsigned long capacity;
    int flags;
    unsigned char * data;
    unsigned char sbuff[UL_DBUFF_SLEN];
};
```

#### Members

len

actual length of stored data

capacity

capacity of allocated buffer

flags

only one flag (UL\_DBUFF\_IS\_STATIC) used now

data

pointer to dynamically allocated buffer

sbuff[UL\_DBUFF\_SLEN]

static buffer for small data sizes

## ul\_dbuff\_init

### Name

`ul_dbuff_init` — init memory allocated for dynamic buffer

### Synopsis

```
int ul_dbuff_init (ul_dbuff_t * buf, int flags);
```

### Arguments

*buf*

buffer structure

*flags*

flags describing behaviour of the buffer only `UL_DBUFF_IS_STATIC` flag is supported. in this case buffer use only static array `sbuf`

### Description

Returns capacity of initialised buffer

## ul\_dbuff\_destroy

### Name

`ul_dbuff_destroy` — frees all resources allocated by `buf`

## Synopsis

```
void ul_dbuff_destroy (ul_dbuff_t * buf);
```

## Arguments

*buf*

buffer structure

## ul\_dbuff\_prep

### Name

`ul_dbuff_prep` — sets a new len and capacity of the buffer

## Synopsis

```
int ul_dbuff_prep (ul_dbuff_t * buf, int new_len);
```

## Arguments

*buf*

buffer structure

*new\_len*

new desired buffer length

## Description

Returns new buffer length

## ul\_dbuff\_set\_capacity

### Name

`ul_dbuff_set_capacity` — change capacity of buffer to at least *new\_capacity*

### Synopsis

```
int ul_dbuff_set_capacity (ul_dbuff_t * buf, int new_capacity);
```

### Arguments

*buf*

buffer structure

*new\_capacity*

new capacity

### Description

Returns real capacity of reallocated buffer

## ul\_dbuff\_set\_len

### Name

`ul_dbuff_set_len` — sets a new len of the buffer, change the capacity if necessary

### Synopsis

```
int ul_dbuff_set_len (ul_dbuff_t * buf, int new_len);
```

## Arguments

*buf*

buffer structure

*new\_len*

new desired buffer length

## Description

Returns new buffer length

# ul\_dbuff\_set

## Name

`ul_dbuff_set` — copies bytes to buffer and change its capacity if necessary like `memset`

## Synopsis

```
int ul_dbuff_set (ul_dbuff_t * buf, byte b, int n);
```

## Arguments

*buf*

buffer structure

*b*

appended bytes

*n*

number of appended bytes

## Returns

length of buffer

# ul\_dbuff\_cpy

## Name

`ul_dbuff_cpy` — copies bytes to buffer and change its capacity if necessary

## Synopsis

```
int ul_dbuff_cpy (ul_dbuff_t * buf, const void * b, int n);
```

## Arguments

*buf*

buffer structure

*b*

appended bytes

*n*

number of appended bytes

## Returns

length of buffer

## ul\_dbuff\_cat

### Name

`ul_dbuff_cat` — appends bytes at end of buffer and change its capacity if necessary

### Synopsis

```
int ul_dbuff_cat (ul_dbuff_t * buf, const void * b, int n);
```

### Arguments

*buf*

buffer structure

*b*

appended bytes

*n*

number of appended bytes

### Returns

length of buffer

## ul\_dbuff\_strcat

### Name

`ul_dbuff_strcat` — appends `str` at the end of buffer and change its capacity if necessary

### Synopsis

```
int ul_dbuff_strcat (ul_dbuff_t * buf, const char * str);
```

## Arguments

*buf*

buffer structure

*str*

string to append

## Description

Returns number length of buffer (including terminating '\0')

# ul\_dbuff\_strcpy

## Name

`ul_dbuff_strcpy` — copy str to the buffer and change its capacity if necessary

## Synopsis

```
int ul_dbuff_strcpy (ul_dbuff_t * buf, const char * str);
```

## Arguments

*buf*

buffer structure

*str*

string to copy



## Description

Returns number length of buffer (including terminating '\0')

## ul\_dbuff\_append\_byte

### Name

`ul_dbuff_append_byte` — appends byte at the end of buffer and change its capacity if necessary

### Synopsis

```
int ul_dbuff_append_byte (ul_dbuff_t * buf, unsigned char b);
```

### Arguments

*buf*

buffer structure

*b*

appended byte

### Description

Returns number length of buffer (including terminating '\0')

## ul\_dbuff\_ltrim

### Name

`ul_dbuff_ltrim` — remove all white space characters from the left

## Synopsis

```
int ul_dbuff_ltrim (ul_dbuff_t * buf);
```

## Arguments

*buf*

buffer structure

## Return

new length of buffer

# ul\_dbuff\_rtrim

## Name

`ul_dbuff_rtrim` — remove all white space characters from the right

## Synopsis

```
int ul_dbuff_rtrim (ul_dbuff_t * buf);
```

## Arguments

*buf*

buffer structure

## Description

if *buf* is terminated by `'\0'`, than is also terminated after `rtrim`

## Return

new length of buffer

## ul\_dbuff\_trim

### Name

`ul_dbuff_trim` — remove all white space characters from the right and from the left

### Synopsis

```
int ul_dbuff_trim (ul_dbuff_t * buf);
```

### Arguments

*buf*

buffer structure

### Description

Returns number length of buffer (including terminating '\0')

## ul\_dbuff\_cpos

### Name

`ul_dbuff_cpos` — searches string for char

## Synopsis

```
int ul_dbuff_cpos (const ul_dbuff_t * buf, unsigned char what, unsigned char quote);
```

## Arguments

*buf*

searched dbuff

*what*

char to find

*quote*

skip str areas quoted in quote chars<br> If you want to ignore quotes assign '\0' to quote in function call

## Return

position of what char or negative value

## ul\_str\_cpos

### Name

`ul_str_cpos` — searches string for char

### Synopsis

```
int ul_str_cpos (const unsigned char * str, unsigned char what, unsigned char quote);
```

## Arguments

*str*

zero terminated string

*what*

char to find

*quote*

skip str areas quoted in quote chars If you want to ignore quotes assign '\0' to quote in function call

## Return

position of what char or negative value

# ul\_str\_pos

## Name

`ul_str_pos` — searches string for substring

## Synopsis

```
int ul_str_pos (const unsigned char * str, const unsigned char * what,  
unsigned char quote);
```

## Arguments

*str*

zero terminated string

*what*

string to find

*quote*

skip str areas quoted in quote chars If you want to ignore quotes assign '\0' to quote in function call

## Return

position of what string or negative value

## ul\_str\_ncpy

### Name

`ul_str_ncpy` — copies string to the buffer

### Synopsis

```
int ul_str_ncpy (unsigned char * to, const unsigned char * from, int
buff_size);
```

### Arguments

*to*

buffer where to copy str

*from*

zero terminated string

*buff\_size*

size of the *to* buffer (including terminating zero)

### Description

Standard `strncpy` function have some disadvantages (ie. do not append term. zero if copied string doesn't fit in to buffer, fills whole rest of buffer with zeros)

Returns `strlen(to)` or negative value in case of error

## ul\_dbuff\_cut\_pos

### Name

`ul_dbuff_cut_pos` — cut first  $n$  bytes from *fromdb* and copies it to *todb*.

### Synopsis

```
void ul_dbuff_cut_pos (ul_dbuff_t * fromdb, ul_dbuff_t * todb, int n);
```

### Arguments

*fromdb*

buffer to cut from

*todb*

buffer to copy to

*n*

position where to cut

### Description

If  $n$  is greater than `fromdb.len` whole *fromdb* is copied to *todb*. If  $n$  is negative position to cut is counted from the end of *fromdb*. If  $n$  is zero *fromdb* stays unchanged and *todb* is resized to len equal zero.

## ul\_dbuff\_cut\_delimited

### Name

`ul_dbuff_cut_delimited` — cuts bytes before delimiter + delimiter char from *fromdb* and copies them to the *todb*

### Synopsis

```
void ul_dbuff_cut_delimited (ul_dbuff_t * fromdb, ul_dbuff_t * todb, char
delimiter, char quote);
```

### Arguments

*fromdb*

buffer to cut from

*todb*

buffer to copy to

*delimiter*

delimiter char

*quote*

quoted delimiters are ignored, *quote* can be `'\0'`, than it is ignored.

### Description

If *fromdb* doesn't contain delimiter *todb* is trimmed to zero length.

## ul\_dbuff\_cut\_token

### Name

`ul_dbuff_cut_token` — cuts not whitespaces from *fromdb* to *todb*.



## Synopsis

```
void ul_dbuff_cut_token (ul_dbuff_t * fromdb, ul_dbuff_t * todb);
```

## Arguments

*fromdb*

buffer to cut from

*todb*

buffer to copy to

## Description

Leading whitespaces are ignored. Cut string is trimmed.

# ul\_dbuff\_export

## Name

`ul_dbuff_export` — Copies data from `srcdb` to the buffer `dest`.

## Synopsis

```
int ul_dbuff_export (ul_dbuff_t * srcdb, void * dest, int maxlen);
```

## Arguments

*srcdb*

source dbuff

*dest*

destination buffer

*maxlen*

maximum number of bytes to copy

## Returns

the number of bytes copied.

## 1.6. Event Connectors

### struct `evc_link`

#### Name

struct `evc_link` — Event Connector Link

#### Synopsis

```
struct evc_link {
    struct src;
    unsigned standalone:1;
} dst;
evc_prop_fnc_t * propagate;
int refcnt;
unsigned recursive:1;
unsigned blocked:1;
unsigned ready:1;
unsigned dead:1;
unsigned delete_pend:1;
unsigned malloced:1;
unsigned standalone:1;
unsigned tx_full_hub:1;
unsigned rx_full_hub:1;
short taken;
};
```

## Members

`src`

describes source of the event link, contains pointer to `&evc_tx_hub_t` and `peers` links list

`standalone`

link is used for standalone function invocation

`dst`

determines destination of the event, it can be `standalone rx_fnc` function with `context` or `&evc_tx_hub_t` in the `multi` case

`propagate`

pointer to the arguments propagation function,

`refcnt`

link reference counter

`recursive`

link can propagate could be invoked recursively, else recursive events are ignored by link

`blocked`

event propagation is blocked for the link, can be used by application

`ready`

link is ready and has purpose to live - it connects two active entities

`dead`

link is dead and cannot propagate events

`delete_pend`

link is being deleted, but it is taken simultaneously, delete has to wait for finish of the propagate and to moving to the next link

`malloced`

link has been malloced and should be automatically freed when referenc counts drop to zero

`standalone`

link is used for standalone function invocation

`tx_full_hub`

`src` points to the full hub structure

`rx_full_hub`

`dst` points to the full hub structure

taken

link is in middle of the propagation process

## Description

The link delivers events from the source to the destination. The link specific function `propagate` is called for each link leading from the hub activated by `evc_tx_hub_emit` and `evc_tx_hub_propagate`. The `propagate` function is responsible for parameters transformation before invocation of standalone or destination hub `rx_fnc` function.

## struct evc\_tx\_hub

### Name

`struct evc_tx_hub` — Event Transmit Hub

### Synopsis

```
struct evc_tx_hub {
    ul_list_head_t links;
};
```

### Members

`links`

list of links outgoing from the hub

## struct evc\_rx\_hub

### Name

`struct evc_rx_hub` — Event Receiving Hub

## Synopsis

```
struct evc_rx_hub {
    ul_list_head_t links;
    evc_rx_fnc_t * rx_fnc;
    void * context;
};
```

## Members

`links`

list of links incoming to the hub

`rx_fnc`

function invoked when event arrives

`context`

context for `rx_fnc`

## evc\_link\_inc\_refcnt

### Name

`evc_link_inc_refcnt` — Increment Link Reference Count

### Synopsis

```
void evc_link_inc_refcnt (evc_link_t * link);
```

### Arguments

*link*

pointer to link

## evc\_link\_dec\_refcnt

### Name

`evc_link_dec_refcnt` — Decrement Link Reference Count

### Synopsis

```
void evc_link_dec_refcnt (evc_link_t * link);
```

### Arguments

*link*

pointer to link

### Description

if the link reference count drops to 0, link is deleted from hubs by `evc_link_dispose` function and if *malloced* is set, link memory is disposed by `free`. Special handlink can be achieved if `propagate` returns non-zero value if called with *ded* link.

## evc\_link\_init

### Name

`evc_link_init` — Initialize Event Connector Link

### Synopsis

```
int evc_link_init (evc_link_t * link);
```

## Arguments

*link*

pointer to the link

## Description

Link reference count is set to 1 by this function

## Return Value

negative value informs about failure.

# evc\_link\_new

## Name

`evc_link_new` — Allocates New Event Connector Link

## Synopsis

```
evc_link_t * evc_link_new ( void );
```

## Arguments

*void*

no arguments

## Description

Link reference count is set to 1 by this function

## Return Value

pointer to the new link or NULL.

## evc\_link\_connect

### Name

`evc_link_connect` — Connects Link between Two Hubs

### Synopsis

```
int evc_link_connect (evc_link_t * link, evc_tx_hub_t * src, evc_rx_hub_t *  
dst, evc_prop_fnc_t * prop);
```

### Arguments

*link*

pointer to the non-connected initialized link

*src*

pointer to the source hub of type `&evc_tx_hub_t`

*dst*

pointer to the destination hub of type `&evc_rx_hub_t`

*prop*

propagation function corresponding to source and destination expected event arguments

### Description

If ready flag is not set, link state is set to ready and reference count is increased.



## Return Value

negative return value indicates fail.

# evc\_link\_init\_standalone

## Name

`evc_link_init_standalone` — Initialize Standalone Link

## Synopsis

```
int evc_link_init_standalone (evc_link_t * link, evc_rx_fnc_t * rx_fnc, void * context);
```

## Arguments

*link*

pointer to the link

*rx\_fnc*

pointer to the function invoked by event reception

*context*

context for the `rx_fnc` function invocation

## Description

Link reference count is set to 1 by this function

## Return Value

negative value informs about failure.

## evc\_link\_new\_standalone

### Name

`evc_link_new_standalone` — Allocates New Standalone Link

### Synopsis

```
evc_link_t * evc_link_new_standalone (evc_rx_fnc_t * rx_fnc, void * context);
```

### Arguments

*rx\_fnc*

callback function invoked if event is delivered

*context*

context provided to the callback function

### Description

Link reference count is set to 1 by this function

### Return Value

pointer to the new link or NULL.

## evc\_link\_connect\_standalone

### Name

`evc_link_connect_standalone` — Connects Standalone Link to Source Hubs

## Synopsis

```
int evc_link_connect_standalone (evc_link_t * link, evc_tx_hub_t * src,
evc_prop_fnc_t * prop);
```

## Arguments

*link*

pointer to the non-connected initialized link

*src*

pointer to the source hub of type &evc\_tx\_hub\_t

*prop*

propagation function corresponding to hub source and standalone `rx_fnc` expected event arguments

## Description

If ready flag is not set, link state is set to ready and reference count is increased.

## Return Value

negative return value indicates failure.

## evc\_link\_delete

### Name

`evc_link_delete` — Deletes Link from Hubs Lists

### Synopsis

```
int evc_link_delete (evc_link_t * link);
```

## Arguments

*link*

pointer to the possibly connected initialized link

## Description

If ready flag is set, link ready flag is cleared and reference count is decreased. This could lead to link disappear, if nobody is holding reference.

## Return Value

positive return value indicates immediate delete, zero return value informs about delayed delete.

# evc\_link\_dispose

## Name

`evc_link_dispose` — Disposes Link

## Synopsis

```
void evc_link_dispose (evc_link_t * link);
```

## Arguments

*link*

pointer to the possibly connected initialized link

## Description

Deletes link from hubs, marks it as dead, calls final death `propagate` for the link and if link is *malloced*, releases link occupied memory.

## evc\_tx\_hub\_init

### Name

`evc_tx_hub_init` — Initializes Event Transmition Hub

### Synopsis

```
int evc_tx_hub_init (evc_tx_hub_t * hub);
```

### Arguments

*hub*

pointer to the `&evc_tx_hub_t` type hub

### Return Value

negative return value indicates failure.

## evc\_tx\_hub\_done

### Name

`evc_tx_hub_done` — Finalize Event Transmition Hub

## Synopsis

```
void evc_tx_hub_done (evc_tx_hub_t * hub);
```

## Arguments

*hub*

pointer to the `&evc_tx_hub_t` type hub

## evc\_tx\_hub\_propagate

### Name

`evc_tx_hub_propagate` — Propagate Event to Links Destinations

## Synopsis

```
void evc_tx_hub_propagate (evc_tx_hub_t * hub, va_list args);
```

## Arguments

*hub*

pointer to the `&evc_tx_hub_t` type hub

*args*

pointer to the variable arguments list

## Description

The function propagates event to the connected links, it skips links marked as *dead*, *blocked* or *delete\_pend*. If the link is not marked as *recursive*, it ensures, that link is not called twice.

## evc\_tx\_hub\_emit

### Name

`evc_tx_hub_emit` — Emits Event to Hub

### Synopsis

```
void evc_tx_hub_emit (evc_tx_hub_t * hub, ...);
```

### Arguments

*hub*

pointer to the `&evc_tx_hub_t` type hub

...

variable arguments

### Description

The function hands over arguments to `evc_tx_hub_propagate` as `&va_list`.

## evc\_rx\_hub\_init

### Name

`evc_rx_hub_init` — Initializes Event Reception Hub

### Synopsis

```
int evc_rx_hub_init (evc_rx_hub_t * hub, evc_rx_fnc_t * rx_fnc, void *  
context);
```

## Arguments

*hub*

pointer to the `&evc_rx_hub_t` type hub

*rx\_fnc*

pointer to the function invoked by event reception

*context*

context for the `rx_fnc` function invocation

## Return Value

negative return value indicates failure.

# evc\_rx\_hub\_disconnect\_all

## Name

`evc_rx_hub_disconnect_all` — Disconnect Reception Hub from All Sources

## Synopsis

```
void evc_rx_hub_disconnect_all (evc_rx_hub_t * hub);
```

## Arguments

*hub*

pointer to the `&evc_rx_hub_t` type hub



## evc\_rx\_hub\_done

### Name

`evc_rx_hub_done` — Finalize Event Reception Hub

### Synopsis

```
void evc_rx_hub_done (evc_rx_hub_t * hub);
```

### Arguments

*hub*

pointer to the `&evc_rx_hub_t` type hub

## 1.7. Application Messages Logging

### struct ul\_log\_domain

#### Name

`struct ul_log_domain` — Loggomg domain structure

#### Synopsis

```
struct ul_log_domain {
    int level;
    const char * name;
    int flags;
};
```

## Members

<code>level</code>	maximal enabled logging level for domain
<code>name</code>	logging domain name
<code>flags</code>	logging domain flags

## ul\_log1

### Name

`ul_log1` — generic logging facility for ULUT library

### Synopsis

```
void ul_log1 (ul_log_domain_t * domain, int level, const char * format,
...);
```

### Arguments

*domain*  
pointer to domain of debugging messages

*level*  
severity level

*format*  
printf style format followed by arguments

...  
variable arguments

## Description

This functions is used for logging of various events. If not overridden by application, logged messages goes to the stderr. Environment variable `UL_LOG_FILENAME` can be used to redirect output to file. Environment variable `UL_DEBUG_FLG` can be used to select different set of logged events through `ul_debug_flg`.

## Note

There is a global variable `ul_log_cutoff_level`. Only the messages with `level <= ul_log_cutoff_level` will be logged.

## ul\_log\_redir

### Name

`ul_log_redir` — redirects default log output function

### Synopsis

```
void ul_log_redir (ul_log_fnc_t * log_fnc, int add_flags);
```

### Arguments

*log\_fnc*

new log output function. Value NULL resets to default function

*add\_flags*

some more flags

## 1.8. Unique IDs Generator

### struct ul\_uniqid\_pool\_t

#### Name

struct ul\_uniqid\_pool\_t — The Unique Identifiers Pool

#### Synopsis

```
struct ul_uniqid_pool_t {
    gavl_fles_int_root_field_t items;
    ul_uniqid_range_t range;
};
```

#### Members

items

GAVL tree of not allocated yet ranges

range

numeric range to allocate IDs from

#### Description

The unique pool provides functions to manage unique numerical IDs. The pool is first initialized by function `ul_uniqid_pool_init`. The available range is specified at this time. The pool can be flushed and destroyed by call `ul_uniqid_pool_done`.

The function `ul_uniqid_pool_alloc_one` returns first free ID from range. The ID is returned to the pool by function `ul_uniqid_pool_free_one`. There are even functions to reserve and release specific IDs range.

## ul\_uniqid\_pool\_init

### Name

`ul_uniqid_pool_init` — Initialize Unique IDs Pool

### Synopsis

```
int ul_uniqid_pool_init (ul_uniqid_pool_t * pool, ul_uniqid_t first,  
ul_uniqid_t last);
```

### Arguments

*pool*

the pointer to the unique IDs pool

*first*

the start of the available numeric range

*last*

the end of the available numeric range

### Return Value

The function returns -1 if the *first* > *last* or if there is not enough memory to allocate item for initial range representation. The zero value indicates successful initialization.

## ul\_uniqid\_pool\_done

### Name

`ul_uniqid_pool_done` — Finalize Unique IDs Pool

## Synopsis

```
int ul_uniqid_pool_done (ul_uniqid_pool_t * pool);
```

## Arguments

*pool*

the pointer to the unique IDs pool

## Return Value

The zero value indicates success.

# ul\_uniqid\_pool\_reserve

## Name

`ul_uniqid_pool_reserve` — Reserve Range from the Unique IDs Pool

## Synopsis

```
int ul_uniqid_pool_reserve (ul_uniqid_pool_t * pool, ul_uniqid_t first,  
ul_uniqid_t last);
```

## Arguments

*pool*

the pointer to the unique IDs pool

*first*

the start value of the range

*last*

the end value of the range

## Description

The function checks if specified range *first..last* is free and reserves it from free pool.

## Return Value

The zero value indicates success. The value of -1 indicates, that range overlaps with already reserved values or exceeds pool boundaries. The value 1 is returned in the case, that there is not enough free memory to represent new non-continuous ranges.

# ul\_uniqid\_pool\_release

## Name

`ul_uniqid_pool_release` — Release Range Back to the Unique IDs Pool

## Synopsis

```
int ul_uniqid_pool_release (ul_uniqid_pool_t * pool, ul_uniqid_t first,  
ul_uniqid_t last);
```

## Arguments

*pool*

the pointer to the unique IDs pool

*first*

the start value of the range

*last*

the end value of the range

## Description

The range *first..last* is returned to the pool for subsequent reuse.

## Return Value

The zero value indicates success. The value of -1 indicates, that range cannot be return back, because there is no free memory to allocate space for returned range.

# ul\_uniqid\_pool\_alloc\_one

## Name

`ul_uniqid_pool_alloc_one` — Allocate/Generate One Unique ID from the Pool

## Synopsis

```
int ul_uniqid_pool_alloc_one (ul_uniqid_pool_t * pool, ul_uniqid_t * ptrid);
```

## Arguments

*pool*

the pointer to the unique IDs pool

*ptrid*

pointer to `ul_uniqid_t` variable where unique ID is returned

## Description

The function allocates lowest available ID from the pool and assigns its value to the space pointed by *ptrid*.



## Return Value

The zero value indicates success. The value of -1 indicates, that all IDs from the pool are taken. No other reason is possible, because function does not call any memory allocation function.

# ul\_uniqid\_pool\_alloc\_one\_after

## Name

`ul_uniqid_pool_alloc_one_after` — Allocate One Unique ID Greater Than Value Specified

## Synopsis

```
int ul_uniqid_pool_alloc_one_after (ul_uniqid_pool_t * pool, ul_uniqid_t *  
ptrid, ul_uniqid_t afterid);
```

## Arguments

*pool*

the pointer to the unique IDs pool

*ptrid*

pointer to `ul_uniqid_t` variable where unique ID is returned

*afterid*

the ID value after which free ID is searched

## Description

The function allocates the first available ID after *afterid* from the pool and assigns its value to the space pointed by *ptrid*. If there is no available ID with value greater than *afterid*, the first free ID from the whole pool is returned.

## Return Value

The zero value indicates success. The value of -1 indicates, that all IDs from the pool are taken. No other reason is possible, because function does not call any memory allocation function.

## ul\_uniqid\_pool\_free\_one

### Name

`ul_uniqid_pool_free_one` — Release One Previously Allocated Unique ID

### Synopsis

```
int ul_uniqid_pool_free_one (ul_uniqid_pool_t * pool, ul_uniqid_t id);
```

### Arguments

*pool*

the pointer to the unique IDs pool

*id*

the released ID value

### Return Value

The zero value indicates success. The value of -1 indicates, that ID cannot be return back, because there is no free memory to allocate space for range representing returned ID.