ADUCID C SDK

Version 3.0.4

Release date 1. February 2016

Antala Staška 2027/79, 140 00 Praha 4 | T +420 271 100 100 | F +420 271 100 101
Vídeňská 204/125, Přízřenice, 619 00 Brno | T +420 547 100 100 | F +420 547 100 101
Sady 5. května 59, 301 00 Plzeň | T +420 271 100 100 | F +420 271 100 101
Teslova 30, 821 02 Bratislava | T +421 (2) 4821 3111 | F +421 (2) 4821 3199

ISO 9001 ISO 14001 ISO/IEC 27001 ISO/IEC 20000

CZ > IČ 25313029 | Registry Court, Regional Court in Brno, ref. no. B2113
SK > IČ 35787546 | Companies Registry, Regional Court in Bratislava, Section Sa, Insert 2431/B

Table of Contents

1. Introduction

This document describes C SDK and integration of third-party applications to enable them to use the ADUCID® technology. It is assumed that the reader has knowledge of web technology, integration of web applications and the C programming language.

2. The installation procedure

2.1. Package installation

C SDK is supplied as an RPM package for RedHat Enterprise Linux 5/6 or CentOS 5/6. Installation in version 6 is executed by the following command:

```
yum localinstall libaducid-[version].el6.noarch.rpm
```

Installation in version 5 is executed by the following command:

```
yum install libxml curl
rpm -ivh libaducid-[version].el5.noarch.rpm
```

Installation of developer libraries for the rhel6 is executed by the following command:

```
yum localinstall libaducid-devel-[version].el6.noarch.rpm
```

Installation for the rhel5 version is executed by the following command:

```
yum install libxml-devel curl-devel
rpm -ivh libaducid-devel-[version].el5.noarch.rpm
```

3. SDK structure

The SDK contains three groups of functions:

- Primitive functions for direct access to the AIM interface
- Higher level functions for ADUCID authentication and PEIG tasks
- Function for auxiliary type tasks: lists, conversion of enumeration types to strings, etc.

4. ADUCID authentication functions

This set of ADUCID functions uses the AducidHandle_t data structure, in which status information is saved.

4.1. The aducid_new function

```
AducidHandle_t aducid_new(char *AIM, char *authId, char *authKey, char *bindingId, char
*bindingKey);
```

The first input parameter is the AIM interface address. It may be provided as DNS name, IP address or URL. Examples of valid parameters:

```
aim.example.com
```

```
1.2.3.4
aim.example.com:80
http://aim.example.com
https://aim.example.com:4443/
http://1.2.3.4
```

Additional parameters include **authId**, **authKey**, **bindingId** and **bindingKey**. The created object saves these parameters and may be used for direct AIM server queries. For unknown parameters, use NULL. When the **authId** value is specified as NULL, all other parameters are ignored!

The **authId** parameter is the ADUCID operation identifier. The **authKey** parameter is the secret created during some authentication scenarios.

The other two parameters are **bindingId** and **bindingKey**. In ADUCID authentication terminology, binding is a method of transferring authentication information between an application communication channel (for example, browser or web server) and an ADUCID communication channel (for example, PEIG or AIM). From the SDK viewpoint, binding is represented by two parameters: **bindingID** and **bindingKey**.

These parameters are created during different types of ADUCID use, in different phases of authentication (depending on the scenario used, PEIG on PC, PEIG on phone; also depending on AIM settings).

For application programmers, the important thing to remember is that as soon as one of these parameters is created, its value must be remembered in the application and the parameter must be used in subsequent calls (binding parameters are used in calls such as **getResult** or **invokePeig**).

The function creates an object in the ADUCID memory and initiates necessary data structures.

The function returns the handle (pointer to a data structure), or NULL when unsuccessful (insufficient memory).

Use **aducid_free()** to terminate and free the memory.

Example of use:

```
#include <aducid.h>;

int main(int argc,char *argv[]) {
    AducidHandle_t handle;

    handle = aducid_new("aim.example.com",NULL,NULL,NULL,NULL);
    // use aducid here
    aducid_free(handle);
}
```

4.2. The aducid_free function

```
extern void aducid_free(AducidHandle_t handle);
```

This function frees the allocated memory and finishes work with the ADUCID object.

4.3. ADUCID identity operations

Generally speaking, the principle of working with identities in ADUCID is as follows: The application asks AIM to initiate an identity operation (for example, 'open', which is an operation that verifies the user's identity). The application obtains the **authId**, or **bindingId** and **bindingKey** identifiers. These identifiers must then be transmitted to the client side and transferred to the user's PEIG (together with the AIM

address). The PEIG then communicates with the AIM and the operation is completed. Depending on the type of operation, the user may be asked to confirm the action.

4.3.1. The aducid_open function

```
char *aducid_open(AducidHandle_t handle, char *peigReturnName);
```

This function initiates the **open** operation on the AIM server. The **open** call is used to use an identity for authentication. The result is the **authId** identifier. The **authId** value is saved in the internal structure and may be referenced later by using **aducid_get_authid**.

The **peigReturnName** parameter is used in web application development. This parameter contains the URL to which the browser is to be redirected after the operation is completed.

The function returns a pointer to **authId**, or, when unsuccessful, NULL (for example, when the AIM server is unavailable).

Example of use:

```
#include <aducid.h>;

int main(int argc,char *argv[]) {
    AducidHandle_t handle;

    handle = aducid_new("aim.example.com",NULL,NULL,NULL,NULL);
    if( aducid_open(handle, NULL) ) {
        // operation has been started
    }
    aducid_free(handle);
}
```

4.3.2. The aducid_init function

```
char *aducid_init(AducidHandle_t handle, char *peigReturnName);
```

This function initiates the **init** operation on the AIM server. The **init** operation is used to create an identity. After **authId** is handed over at PEIG, PEIG starts to communicate with the AIM server and after the user confirms, an identity is created (both at AIM and PEIG).

The function returns a pointer to **authId**, or, when unsuccessful, NULL.

4.3.3. The aducid_reinit function

```
char *aducid_reinit(AducidHandle_t handle, char *peigReturnName);
```

This function initiates the **reinit** operation on the AIM server. The **reinit** operation is used to renew an identity and recover from an error condition when an identity exists in PEIG, but not on the AIM server (for example, after deletion by the AIM server administrator).

The function returns a pointer to **authId**, or, when unsuccessful, NULL.

4.3.4. The aducid_change function

```
char *aducid_change(AducidHandle_t handle, char *peigReturnName);
```

This function initiates the **change** operation on the AIM server. The **change** operation is used to change an identity (create new pseudo random identifiers) during its validity. If the identity's validity has expired, use **rechange**. With standard AIM settings, identities are changed automatically before they expire.

The function returns a pointer to **authId**, or, when unsuccessful, NULL.

4.3.5. The aducid_rechange function

```
char *aducid_rechange(AducidHandle_t handle, char *peigReturnName);
```

This function initiates the **rechange** operation on the AIM server. The **rechange** operation is used to change an identity (create new pseudo random identifiers) after its validity has expired. An identity's validity depends both on the number of its uses and on its validity period. The **rechange** operation may be necessary when a user has not logged in to a given service for a long time.

The function returns a pointer to **authId**, or, when unsuccessful, NULL.

4.3.6. The aducid_delete function

```
char *aducid_delete(AducidHandle_t handle, char *peigReturnName);
```

This function initiates the **delete** operation on the AIM server. The **delete** operation is used to remove an identity both from AIM and PEIG.

The function returns a pointer to **authId**, or, when unsuccessful, NULL.

4.3.7. The aducid_exuse function

```
const char* aducid_exuse(
    AducidHandle_t handle,
    const char * methodName,
    const AducidAttributeList_t methodParameters,
    const AducidAttributeList_t personalObject,
    const char * peigReturnURL
);
```

This function initiates the exuse operation on the AIM server. This function allows use advanced ADUCID features like transaction. Because direct usage of this method is complicated, SDK offers other high-level functions, which encapsulate this one. Consult ADUCID generic documentation for details. Several examples can be found also in aducid testing application. Take a look at testing-app.ecpp file for inspiration. Here is list of functions which uses “exuse” operation

- aducid_init_personal_factor
- aducid_change_personal_actor
- aducid_delete_personal_factor
- aducid_verify_personal_factor
- aducid_init_payment
- aducid_confirm_text_transaction
- aducid_confirm_money_transaction
- aducid_create_room_by_name
- aducid_eneter_room_by_name
- aducid_create_room_by_story
- aducid_eneter_room_by_story
- aducid_peig_local_link

For evaluating transaction operation there is aducid_verify_transaction function.

4.4. The aducid_close function

```
bool aducid_close(AducidHandle_t handle);
```

This function closes the ADUCID session on the AIM server. This function should be called when the application obtains all necessary information (authentication result, user information) and will not need ADUCID any more. When the function is not called, the session is automatically closed on the server after a time limit expires. The time limit depends on the AIM server configuration.

The function returns **`true`** when the call is successful.

4.5. The aducid_get_psl_attributes function

```
AducidAIMGetPSLAttributesResponse *aducid_get_psl_attributes(
    AducidHandle_t handle,
    AducidAttributeSet attributeSet,
    bool useCache
);
```

ADUCID offers several attributes directly related to ADUCID identification of the user and their PEIG. This is called Permanent Secure Link (PSL), in the ADUCID terminology. PSL contains selected information about identities and the status of the authentication process.

The call parameters are as follows:

- `handle`
- `attributeSet`—PSL attribute set
- `useCache`—allows reading of required data from cache

The **`attributeSet`** parameter may have the following values:

Set	Meaning
ADUCID_ATTRIBUTE_SET_INVALID	Wrong attribute set/SDK error
ADUCID_ATTRIBUTE_SET_STATUS	Authentication process status
ADUCID_ATTRIBUTE_SET_BASIC	Basic information set
ADUCID_ATTRIBUTE_SET_ALL	Complete information set
ADUCID_ATTRIBUTE_SET_VALIDITY	Identity validity information
ADUCID_ATTRIBUTE_SET_LINK	Information necessary to use identity link
ADUCID_ATTRIBUTE_SET_ERROR	Description of the ADUCID operation's error code
ADUCID_ATTRIBUTE_PEIG_RETURN_NAME	Return URL for applications

4.6. The aducid_ wait_for_operation function

```
bool aducid_wait_for_operation(AducidHandle_t handle);
```

This function waits for an operation to close. The function returns **`true`** when the operation is closed; however, that does not necessarily mean that the operation was successful. When the function returns **`false`**, it indicates that the operation's condition could not be determined. That may happen during a network failure, for instance.

Internally, the function repeatedly calls **`aducid_get_psl_attributes`** to determine the authentication status.

```
aducid_get_psl_attributes(
    handle,
    ADUCID_ATTRIBUTE_SET_STATUS,
    false
);
```

This call is repeated as long as the return value is **statusAIM ADUCID_AIM_STATUS_START** or **ADUCID_AIM_STATUS_WORKING**.

4.7. The aducid_aducid_verify function

```
bool aducid_verify(AducidHandle_t handle);
```

This function determines whether the operation was successful and the user was authenticated. At the time of calling this function, the **authId** and **authKey** parameters must be known.

```
h = aducid_new("aim.example.com",authId,authKey,bindingId,bindingKey);
if( aducid_verify(h) ) {
    // ok
} else {
    // failed
}
aducid_free(h);
```

4.7.1. Note concerning authKey2

AIM settings may specify that **authKey** is to be used once only. Under such settings, AIM returns **authKey2** when the first **aducid_get_psl_attributes** call is made. From that moment on, this new secret is to be used.

The **aducid_get_psl_attributes** function automatically saves **authKey2** into the **AducidHandle_t** internal data structure, instead of the old **authKey**.

4.8. The aducid_get_attributes function

```
AducidAttributeList_t *aducid_get_attributes(AducidHandle_t handle,char *attrSetName);
```

This function is an equivalent of the use of the **AIMexecutePersonalObject** low-level call, with the **Read** method, **attrSetName** as the name of personal object and **USER_ATTRIBUTE_SET** as the algorithm. Because the **AIMexecutePersonalObject** call is general and enables many operations with user data saved with the provider, the ADUCID SDK offers simplified calls for typical operations. One of those is reading and writing of attributes saved in the database of users.

ADUCID enables attribute sets to be created. Sets include list of attributes, their mapping to LDAP and access rights for each attribute (read, write). The rights here are a protection against programmer errors, because ADUCID does not prevent the application to use any of the sets defined. The pre-defined sets are "`default`" and "UIM". Individual attributes available from ADUCID are fully configurable and depend on deployment conditions.

The **aducid_get_attributes** function returns a list of authenticated user's attributes, or NULL when unsuccessful. The attribute list must be cleared by using **aducid_attr_list_free()**.

```
AducidAttributeList_t *list;
char *surname, *givenname;
AducidHandle_t h;

h = aducid_new("aim.example.com",authId,authKey,bindingId,bindingKey);
```

```
if( aducid_verify(h) ) {
    // ok
    list = aducid_get_attributes(h, "default");
    surname = aducid_attr_list_get_first_by_name(list, "sn");
    givenname = aducid_attr_list_get_first_by_name(list, "givenName");
    printf("User: %s %s\n",givenname,surname);
    aducid_attr_list_free(list);
}
aducid_free(h);
```

4.9. The aducid_set_attributes function

```
bool aducid_set_attributes(
    AducidHandle_t handle,
    char *attrSetName,
    AducidAttributeList_t *attrs
);
```

This function writes user attributes into the ADUCID database. Its use is similar to that of the **aducid_get_attributes** function. The list of attributes being written does not have to be a complete set (and cannot be, when some attributes are read-only). The list must contain only attributes which are being changed. The function returns **true** when the operation is successful.

```
AducidAttributeList_t *list;
char *surname, *givenname;
AducidHandle_t h;

h = aducid_new("aim.example.com",authId,authKey,bindingId,bindingKey);
if( aducid_verify(h) ) {
    list = aducid_attr_list_new();
    aducid_attr_list_append(list, "sn","Smith");
    aducid_attr_list_append(list, "givenName","John");
    if( ! aducid_set_attributes(list, "UIM",list) ) {
        printf("error: writing attributes\n");
    }
    aducid_attr_list_free(list);
}
aducid_free(h);
```

4.10. The aducid_get_user_database_index function

```
char *aducid_get_user_database_index(AducidHandle_t handle);
```

The PSL User Database Index (UDI) is very interesting from the viewpoint of applications that are linked to ADUCID. It is a meaningless user identifier, constant over time. UDI is identical even if the user uses their back-up PEIG, created with a replica.

The function returns (char *) pointer, or, when unsuccessful, NULL.

```
AducidHandle_t h;

h = aducid_new("aim.example.com",authId,authKey,bindingId,bindingKey);
if( aducid_verify(h) ) {
    printf("user database index: %s\n", aducid_get_user_database_index(h) );
}
aducid_free(h);
```

4.11. The aducid_clear_psl_cache function

```
void aducid_clear_psl_cache(AducidHandle_t handle);
```

The ADUCID SDK stores information obtained from PSL in a cache memory, to prevent repeated network queries. This memory may be cleared by using the **aducid_clear_psl_**cache function. However, this is not necessary, as the memory is automatically cleared when the **`aducid_free()`** call is made.

4.12. The aducid_get_authid function

```
char *aducid_get_authid(AducidHandle_t handle);
```

This function returns a pointer to **`authId`**, or NULL, when the **`authId`** is not known.

4.13. The aducid_set_authid function

```
void aducid_set_authid(AducidHandle_t handle, char *authId);
```

This function sets the **`authId`** for the given handle.

4.14. The aducid_get_authkey function

```
char *aducid_get_authkey(AducidHandle_t handle);
```

The function returns a pointer to **`authKey`**, or NULL when the **`authKey`** is not known.

4.15. The aducid_set_authkey function

```
void aducid_set_authkey(AducidHandle_t handle, char *authKey);
```

This function sets the **`authKey`** for the given handle.

4.16. The aducid_get_bindingid function

```
char *aducid_get_bindingid(AducidHandle_t handle);
```

The function returns a pointer to **`bindingId`**, or NULL when the **`bindingId`** is not known.

4.17. The aducid_set_bindingid function

```
void aducid_set_bindingid(AducidHandle_t handle, char *bindingId);
```

This function sets the **`bindingId`** for the given handle.

4.18. The aducid_get_bindingkey function

```
char *aducid_get_bindingkey(AducidHandle_t handle);
```

The function returns a pointer to **`bindingKey`**, or NULL when the **`bindingKey`** is not known.

4.19. The aducid_set_bindingkey function

```
void aducid_set_bindingkey(AducidHandle_t handle, char *bindingKey);
```

This function sets the **bindingKey** for the given handle.

4.20. Communication with PEIG

The library contains several functions enabling communication with PEIG.

4.20.1. The aducid_peig_invoke function

```
bool  aducid_peig_invoke(AducidHandle_t handle);
```

This function may be used when creating a thick client in a client-server architecture. Its use with web applications is not beneficial. The function communicates with a PEIG-proxy through a redirect adapter (http connection to **localhost**).

Calling the function transfers parameters for initialisation of the corresponding ADUCID operation to the PEIG/PEIG-proxy.

The function returns **true** when the information transfer is successful.

```
AducidHandle h;

h = aducid_new("aim.example.com",authId,NULL,bindingId,bindingKey);
aducid_peig_invoke(h);
aducid_free(h);
```

4.20.2. The aducid_peig_get_authkey function

```
char *aducid_peig_get_authkey(AducidHandle_t handle);
```

This function is also intended for use with a thick client and is supplementary to **aducid_peig_invoke**. It is used to obtain the **authKey**—the resulting secret that the client may use to authenticate themselves.

```
AducidHandle_t handle;
char *URL;

// we received authId, bindingId, bindingKey from server
handle = aducid_new(TESTINGAIM,authId,NULL,bindingId,bindingKey);
// let the peig know
if( aducid_peig_invoke(handle) ) {
    // peig started (or maybe QR code has been displayed)
    if( aducid_wait_for_operation(handle) ) {
        // operation finished
        aducid_peig_get_authkey(handle);
        if( aducid_verify(handle) ) {
            // user authenicated
            printf("ok\n");
        } else {
            printf("verification failed\n");
        }
    } else {
        printf("wait failed (network issue?)\n");
    }
}
if( ! aducid_close(handle) ) {
```

```
    printf("close ADUCID failed\n");
};
aducid_free(handle);
```

4.20.3. The aducid_get_aimproxy_url function

```
char *aducid_get_aimproxy_url(AducidHandle handle);
```

This function generates a URL which may be used to redirect the user to the AIM-proxy. The resulting string must be cleared by using **free()**.

Examples of URLs generated in this way:

```
http://aim.example.com/AIM-proxy/process?authId=w3P31ufKiug%3D&
bindingId=5xW6xo%2F25jw%3D&bindingKey=nOjglEfyewc%3D
```

4.20.4. The aducid_get_aducid_url function

```
char *aducid_get_aducid_url(AducidHandle_t handle);
```

This function generates a URL for an ADUCID diagram. This URL may be used to generate a QR code for use in mobile PEIG authentication. The resulting string must be cleared by using **free()**.

Examples of URLs generated in this way:

```
aducid://callback?authId=w3P31ufKiug%3D&
r3Url=http%3A%2F%2Faim.example.com%2FAIM%2Fservices%2FR3&
bindingId=5xW6xo%2F25jw%3D&bindingKey=nOjglEfyewc%3D
```

5. Functions for operations with attribute lists

The **aducid_get_attributes** and **aducid_set_attributes** functions work with a generic list of user data, such as name, surname and email. Each attribute has its name and value. Both name and value are char * type. Items are indexed starting from zero.

5.1. The aducid_attr_list_new function

```
AducidAttributeList_t aducid_attr_list_new();
```

This function creates a new empty attribute list and returns its handle. This list and its contents must be cleared by using **aducid_attr_list_free**.

5.2. The aducid_attr_list_free function

```
void aducid_attr_list_free(AducidAttributeList_t handle);
```

This function clears a list of attributes from the memory.

5.3. The aducid_attr_list_count function

```
int aducid_attr_list_count(AducidAttributeList_t handle);
```

This function returns the number of items in a list.

Example of use:

```
if( aducid_attr_list_count(handle) ) {
    printf("list is not empty\n");
}
```

5.4. The aducid_attr_list_get_item_name function

```
char *aducid_attr_list_get_item_name(AducidAttributeList_t handle,int idx);
```

This function returns the name of the **n** item of the list, or NULL when the given index is invalid.

5.5. The aducid_attr_list_get_item_value function

```
char *aducid_attr_list_get_item_value(AducidAttributeList_t handle,int idx);
```

This function returns the name of the **n** item of the list, or NULL when the given index is invalid.

```
int a;
for(a=0; a<aducid_attr_list_count(handle), a++ ) {
    printf("name[%i] is %s\nvalue[%i] is %s\n",
        a,
        aducid_attr_list_get_item_name(handle,a),
        a,
        aducid_attr_list_get_item_value(handle,a)
    );
}
```

5.6. The aducid_attr_list_append function

```
void aducid_attr_list_append(AducidAttributeList_t handle,char *name,char *value);
```

This function adds a new attribute to the end of the list. Both its name and value parameters must not be NULL. The function creates a copy of the transferred values in the memory (strdup).

5.7. The aducid_attr_list_prepend function

```
void  aducid_attr_list_prepend(AducidAttributeList_t handle,char *name,char *value);
```

This function has the same properties as **aducid_attr_list_appendm**, but adds the new item to the beginning of the list.

5.8. The aducid_attr_list_insert function

```
void aducid_attr_list_insert(AducidAttributeList_t handle,char *name,char *value,int idx);
```

This function inserts an attribute to a given position in the list. When the index points to beyond the list (even if the position is nonsense), the function behaves as **aducid_attr_list_append**. When the index is smaller than 1, the function behaves as **aducid_attr_list_prepend**.

```
AducidAttributeList_t handle;

handle = aducid_attr_list_new();
aducid_attr_list_append(handle,  "item 3","good evening");
aducid_attr_list_prepend(handle, "item 1","good moorning");
aducid_attr_list_insert(handle,  "item 2","good afternoon",1);
// the list is now correctly sorted
aducid_attr_list_free(handle);
```

5.9. The aducid_attr_list_get_first_by_name function

```
char *aducid_attr_list_get_first_by_name(AducidAttributeList_t handle,char *name);
```

This function finds the given attribute and returns a pointer with its value. The search is not case sensitive. A list may contain several attributes of the same name. This is necessary for operations with multi-value attributes in LDAP, such as email addresses. A user may have several email addresses. In such cases, attribute lists contain several "mail" items with different values.

The function also sets the internal pointer and the next **aducid_attr_list_get_next_by_name** call points to the next value.

When the list does not contain an attribute of the given name, the function returns NULL

5.10. The aducid_attr_list_get_next_by_name function

```
char *aducid_attr_list_get_next_by_name(AducidAttributeList_t handle,char *name);
```

The function searches for the next attribute of the given name.

```
// prints all mails on the list
char *mail;

mail = aducid_attr_list_get_first_by_name(handle, "mail");
while(mail) {
    printf("mail: %s\n", mail);
    mail = aducid_attr_list_get_next_by_name(handle, "mail");
}
```

5.11. The aducid_attr_list_get_count_by_name function

```
int aducid_attr_list_get_count_by_name(AducidAttributeList_t handle,char *name);
```

The function counts the attributes of the given name and it is not case sensitive.

```
AducidAttributeList_t handle;
int cnt;

handle = aducid_attr_list_new();
aducid_attr_list_append(handle,  "item","good evening");
aducid_attr_list_append(handle,  "ITEM","good moorning");
cnt = aducid_attr_list_get_count_by_name(handle, "iTEm");
// cnt == 2
```

```
aducid_attr_list_free(handle);
```

5.12. The aducid_attr_list_delete function

```
bool aducid_attr_list_delete(AducidAttributeList_t handle,int idx);
```

This function deletes the item at position **idx** from the list of attributes. When the operation is successful, the function returns **true**. When the function returns **false**, **idx** points to outside of the list, or handle is NULL.

5.13. The aducid_attr_list_delete_by_name function

```
bool  aducid_attr_list_delete_by_name(AducidAttributeList_t handle,char *name);
```

This function deletes all attributes of the given name from the list. The search is not case sensitive.

6. Low-level ADUCID call functions

Low-level functions are used to call the R4 ADUCID interface, by using the SOAP protocol. For example, by creating an XML SOAP query, parsing of the result and http communication with the AIM server.

Names of all functions from this group start with a prefix of **aducid_free_aim_** and **aducid_aim_.**

6.1. The aducid_free_aim_* function

```
void aducid_free_aim_request_operation_response(
    AducidAIMRequestOperationResponse *response
);
void aducid_free_aim_get_psl_attributes_response(
    AducidAIMGetPSLAttributesResponse *response
);
void aducid_free_aim_execute_personal_object_response(
    AducidAIMExecutePersonalObjectResponse *response
);
```

These functions are intended to clear memory structures obtained with low-level functions. The functions test whether the transferred parameter has a response value of NULL; if it does, the function does nothing.

6.2. The aducid_aim_request_operation function

```
AducidAIMRequestOperationResponse *aducid_aim_request_operation(
    char *R4,
    AducidOperation operation,
    char *AIMName,
    char *authId,
    char *methodName,
    char *methodParameter
    char *personalObject,
    char *AAIM2,
    char *ilData,
    char *peigReturnName
);
```

This function initiates an operation on the AIM server. The parameters are as follows:

- `R4`—address of the SOAP R4 AIM interface (for example, "http://aim.example.com/AIM/services/R4")
- `operation`—required operation (see Chapter 10.1 AducidOperation)
- `authId`—ADUCID session identifier (when the value is NULL, it is assigned to AIM)
- `methodName` —not currently used; should have a value of NULL
- `methodParametr` —not currently used; should have a value of NULL
- `personalObject` —not currently used; should have a value of NULL
- `AAIM2`—secondary AIM address for identity link
- `ilData`—data for identity link
- `peigReturnName`—URL for application return address

The function returns a pointer to the **`AducidAIMRequestOperationResponse`** data structure.

```
typedef struct {
    char *authId;
    char *bindingId;
    char *bindingKey;
    AducidAIMStatus statusAIM;
    AducidAuthStatus statusAuth;
} AducidAIMRequestOperationResponse;
```

The response must be cleared by using **`aducid_free_aim_request_operation_response()`**.

6.3. The aducid_aim_get_psl_attributes function

```
AducidAIMGetPSLAttributesResponse *aducid_aim_get_psl_attributes(
    char *R4,
    char *authId,
    char *bindingId,
    char *AIMName,
    char *authKey,
    AducidAttributeSet attributeSet
);
```

The function returns a pointer to the **`AducidAIMGetPSLAttributesResponse`**, or NULL when unsuccessful.

```
typedef struct {
    AducidAIMStatus statusAIM;
    AducidAuthStatus statusAuth;
    AducidOperation operationName;
    char *userDatabaseIndex;
    char *userId;
    char *validityCount;
    char *validityTime;
    char *orangeCount;
    char *orangeTime;
    char *securityProfileName;
    char *authenticationProtocolName;
    char *securityLevel;
    char *ILID;
    char *ilTypeName;
    char *ilAlgorithmName;
    char *ilValidityCount;
    char *ilValidityTime;
    char *authKey2;
    char *sessionKey;
    char *bindingType;
} AducidAIMGetPSLAttributesResponse;
```

The return structure has various items completed, depending on the queried **attributeSet**. Its typical use is a query for a set of attributes **ADUCID_ATTRIBUTE_SET_STATUS**. This set returns the authentication process status (initiated, in progress, completed, etc.) and does not need **authKey** (**authKey** = NULL).

After authentication is completed, other sets may be queried. For normal applications, the **ADUCID_ATTRIBUTE_SET_ALL** set may be used. It contains all necessary information, such as authentication status and user database index.

The structure must be cleared by using **aducid_free_aim_get_psl_attributes_response()**.

6.4. The aducid_aim_execute_personal_object function

```
AducidAIMExecutePersonalObjectResponse *aducid_aim_execute_personal_object(
    char *R4,
    char *authId,
    char *AIMName,
    char *authKey,
    AducidAIMPersonalObjectMethod methodName,
    char *personalObjectName,
    AducidAIMPersonalObjectAlgorithm personalObjectAlgorithm,
    AducidAttributeList_t *personalObjectData,
    char *ILID,
    char *AAIM2,
    char *ilData
);
```

The input parameters are:

- R4—address of the SOAP R4 AIM interface (e.g. "http://aim.example.com/AIM/services/R4")
- authId—ADUCID session identifier
- AIMName—not currently used; should be NULL
- authKey—authentication secret
- methodName—name of the method called (for example, **ADUCID_AIM_PERSONAL_OBJECT_METHOD_WRITE**)
- personalObjectName—object name (for example, name of set being written)
- personalObjectAlgorithm—algorithm used (for example, **ADUCID_AIM_PERSONAL_OBJECT_ALGORITHM_USER_ATTRIBUTE_SET**)
- personalObjectData—list of attributes and their values (for example, name of set being written)
- ILID—information for identity link
- AAIM2—secondary AIM for identity link
- ilData—data for identity link

The function returns a pointer to **AducidAIMExecutePersonalObjectResponse**, or NULL when unsuccessful.

```
typedef struct {
    AducidAIMStatus statusAIM;
    AducidAuthStatus statusAuth;
    char *statusMessage;
    AducidAttributeList_t personalObject;
} AducidAIMExecutePersonalObjectResponse;
```

The structure must be cleared by using **aducid_free_aim_execute_personal_object_response()**.

6.5. The aducid_aim_close_session function

This function closes the authentication session on the AIM server. For security reasons, this function should be called when the authentication session is not needed. If the session is not closed explicitly, it closes itself after a time limit elapses (the limit is configured on the AIM server).

The function returns **`true`** when the call is successful.

7. Functions enabling conversion of enumeration types to strings and vice versa

```
const char *aducid_operation_str(AducidOperation operation);
AducidOperation aducid_operation_enum(char *operation);

const char *       aducid_attribute_set_str(AducidAttributeSet set);
AducidAttributeSet aducid_attribute_set_enum(char *set);

const char *    aducid_aim_status_str(AducidAIMStatus status);
AducidAIMStatus aducid_aim_status_enum(char *status);

const char *
aducid_aimpersonal_object_method_str(AducidAIMPersonalObjectMethod method);
AducidAIMPersonalObjectMethod aducid_aimpersonal_object_method_enum(char *method);

const char *
aducid_aim_personal_object_algorithm_str(AducidAIMPersonalObjectAlgorithm alg);
AducidAIMPersonalObjectAlgorithm aducid_aim_personal_object_algorithm_enum(char *alg);

const char *     aducid_auth_status_str(AducidAuthStatus status);
AducidAuthStatus aducid_auth_status_enum(char *status);
```

For the purposes of text representation of enumeration types (for example, creation of SOAP queries, logs), the ADUCID SDK contains functions enabling conversion of enumeration types to strings and vice versa. When a non-existent name is converted to an enumeration type, the function returns a value of 0 (*_INVALID).

```
AducidOperation operation;

operation = aducid_operation_enum("non existing operation");
// operation == ADUCID_OPERATION_INVALID
// (int)operation == 0
operation = aducid_operation_enum("open");
// operation == ADUCID_OPERATION_OPEN
```

When a non-existent enumeration type value is converted to text, the function returns NULL.

```
char *operation;

operation = aducid_operation_str( (AducidOperation)(-1) );
// operation == NULL
operation = aducid_operation_str(ADUCID_OPERATION_OPEN);
// operation -> "open"
```

8. Library initialisation and de-initialisation

```
int aducid_library_init();
int aducid_library_free();
```

The library contains the **aducid_library_init()** function, which initialises the libraries used (on Linux cURL) and the **aducid_library_free()** which clears them. The **aducid_library_init()** function should be called immediately after the program launches. If the function is not called, the library may not function correctly in multi-thread applications.

The function returns 0 when the call is successful and an error code when it is not (on Linux this is the cURL initialisation error code).

9. C++ binding

There is also C++ class AducidClient. The AducidClient class encapsulates C methods – for example aducid_init function corresponds to AducidClient::init. Also some structrures are converted into stl classes.The aducid_attr_list is converted into std::vector or std::map.

The AducidClient also takes care of authKey2 as well as memory de/allocation and encapsulates exuse method into several high-level methods.

Here is part of the class declaration just for illustration:

```
namespace aducid {
    class AducidClient {
    public:
        AducidClient( const string &AIM );
        AducidClient( const char *AIM );

        string authId() const;
        void authId( const char *authId );
        string authKey() const;
        void authKey( const char *authKey );
…
        bool open( const char *peigReturnURL );
        bool init( const char *peigReturnURL );
        bool change( const char *peigReturnURL );
…
        bool initPersonalFactor( const char *peigReturnURL );
        bool changePersonalFactor( const char *peigReturnURL );
        bool deletePersonalFactor( const char *peigReturnURL );
        bool verifyPersonalFactor( const char *peigReturnURL );
…
    }
}
```

10. Complete documentation is part of the source code in doxygen format. Enumeration types

10.1. AducidOperation

The type defines operations with identity.

Operation	Meaning
ADUCID_OPERATION_INVALID	Wrong operation, SDK error or non-compatibility.
ADUCID_OPERATION_OPEN	The open operation—use of identity. This operation is intended for regular authentication.
ADUCID_OPERATION_INIT	Creates an identity. This operation must be used when a user demonstrably does not have an identity, for example, as a reaction to the USP error code.

ADUCID_OPERATION_REINIT	Renewal of identity. This operation is needed when an identity exists on PEIG, but not on the server (for example, it was deleted by an administrator). Reinit must be used when authentication returns the UU error code.
ADUCID_OPERATION_CHANGE	Change of identity. During this operation, new pseudo-random identifiers are generated for a valid identity.
ADUCID_OPERATION_RECHANGE	Change of identity. During this operation, new pseudo-random identifiers are generated for an invalid identity (for example, an expired one).
ADUCID_OPERATION_DELETE	Identity delete. The identity is deleted both from PEIG and from the server.
ADUCID_OPERATION_REPLICA	PEIG back-ups/replicas are created.
ADUCID_OPERATION_LINK	Identity link—transfer of user information between two AIM servers.
ADUCID_OPERATION_EXUSE	Extended USE—see general ADUCID documentation.

Table 10-1 Identity operations

10.2. AducidAttributeSet

```
typedef enum {
    ADUCID_ATTRIBUTE_SET_INVALID = 0,
    ADUCID_ATTRIBUTE_SET_STATUS,
    ADUCID_ATTRIBUTE_SET_BASIC,
    ADUCID_ATTRIBUTE_SET_ALL,
    ADUCID_ATTRIBUTE_SET_VALIDITY,
    ADUCID_ATTRIBUTE_SET_LINK,
    ADUCID_ATTRIBUTE_SET_ERROR,
    ADUCID_ATTRIBUTE_SET_PEIG_RETURN_NAME
} AducidAttributeSet;
```

10.3. AducidAIMStatus

The following table lists the semantics of individual statuses of the authentication session (AIMStatus): All statuses have a prefix of **ADUCID_AIM_STATUS_**, for example **ADUCID_AIM_STATUS_NONE** and **ADUCID_AIM_STATUS_START**.

Status	Brief description	Note
NONE	Initial status	Initial status of the status diagram.
START	Status that occurrs when the authentication session is generated on AIM	From this point, it is possible to inquire about the status of the authentication process using authId.
START_TIMEOUT	Final status - PEIG did not start communicating within the defined time frame	This status typically means that there is a problem either in communication between PEIG/AIM, or that PEIG-Proxy is not available.
WORKING	Authentication handshake between PEIG and AIM was successfully initiated.	This status means that the authentication handshake has begun.
PROCESS_TIMEOUT	Final status - the authentication handshake was not performed within the defined time frime	This status typically means that the responses from PEIG are slow or the user's reaction when approving a request is slow.

Status	Brief description	Note
ERROR	Final status - problem during authentication	"Masked" problem during authentication. The specific reason can be obtained by a subsequent query via the AIMGetPSLAttributes command of the R4 interface with attributeSet="Error". A list of possible return statuses is provided in the **Chyba! Nenalezen zdroj odkazů.** section.
FINISHED	Authentication handshake was successful	The authentication process was performed without problems. The operation result is available on the server. The operation result can be obtained using a pair (authId, authKey).
ACTIVE	Authentication session was verified	Verification was performed and the authentication session is prepared to work for a period of time defined by the system.
PASSIVE	Passive status	Used on the secondary AIM during the identity link operation.
END	Authentication session ended	The authentication session has been terminated by the program, or the time period defined for work with an open session has expired.
INTERNAL_ERROR	Undefined error	Unspecified error.
AUTH_ERROR	Error in authentication secret	The authentication handshake ended and the valid authentication secret authKey is not used.
CLIENT_BINDING	Binding is executed	The communication before the start of an operation enforcing the binding mode and generating the needed information

Table 10-2: Semantics of individual statuses of the authentication session (AIMStatus)

10.4. AducidAIMPersonalObjectMethod

```
typedef enum {
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_INVALID = 0,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_CREATE,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_CHANGE,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_READ,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_DELETE,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_WRITE,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_LEGACY_LOGIN,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_SET,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_GEN_KEY_PAIR,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_WRITE_CERT,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_LEGACY_SIGN,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_OTP_VERIFICATION,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_IDENTITY_LINK_READ,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_PASSWORD_DECRYPT,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_SET_EXECUTE_RIGHT,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_CLEAR_EXECUTE_RIGHT,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_UPLOAD_CERT,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_TEMP_ENABLE_CAPI,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_TEMP_DISABLE_CAPI,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_PERM_ENABLE_CAPI,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_PERM_DISABLE_CAPI,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_PPO_RESULT,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_AIM_READ,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_READ_PEIG_COUNT,
```

```
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_ACTIVATE_OTHER_PEIGS,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_DEACTIVATE_OTHER_PEIGS,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_ACTIVATE_THE_PEIG,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_DEACTIVATE_THE_PEIG,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_READ_PEIG_ID,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_READ_OTHER_PEIGS_ID,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_CREATE_ROOM_BY_NAME,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_CREATE_ROOM_BY_STORY,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_ENTER_ROOM_BY_NAME,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_ENTER_ROOM_BY_STORY,
    ADUCID_AIM_PERSONAL_OBJECT_METHOD_CONFIRM_TRANSACTION
} AducidAIMPersonalObjectMethod;
```

10.5. AducidAIMPersonalObjectAlgorithm

```
typedef enum {
    ADUCID_AIM_PERSONAL_OBJECT_ALGORITHM_INVALID = 0,
    ADUCID_AIM_PERSONAL_OBJECT_ALGORITHM_USER_ATTR_SET,
    ADUCID_AIM_PERSONAL_OBJECT_ALGORITHM_PASSWD,
    ADUCID_AIM_PERSONAL_OBJECT_ALGORITHM_PASSWD_AIM,
    ADUCID_AIM_PERSONAL_OBJECT_ALGORITHM_BIN_STRING,
    ADUCID_AIM_PERSONAL_OBJECT_ALGORITHM_OTP_AIM_1,
    ADUCID_AIM_PERSONAL_OBJECT_ALGORITHM_X509_SIG,
    ADUCID_AIM_PERSONAL_OBJECT_ALGORITHM_PAYMENT
} AducidAIMPersonalObjectAlgorithm;
```

10.6. AducidAuthStatus

This appendix provides a complete list of potential results of operations within the authentication process framework. All codes in this SDK have a prefix of **ADUCID_AUTHSTATUS_**, for example, **ADUCID_AUTHSTATUS_OK**, **ADUCID_AUTHSTATUS_KO**, **ADUCID_AUTHSTATUS_NAU**.

Code	Brief description	Note
OK	Success	Normal → normal success
KO	Failure	Normal → normal negative result
NAU	Disagreement at user side	Rejected by user - real or artificial because PEIG is temporarily blocked → normal authentication failure
PPNP	PEIG is not available	Functional problem - PEIG is not connected to PEIG-Proxy → Display to user (with instructions)
USP	Unknown service provider	Normal status - unknown user → management (initialization menu - init)
USSP	Unknown secondary service provider	Normal status - unknown user for secondary provider → management (explanatory text - link with other SP)
UU	Unknown user	Operating status or security problem - the identity obtained from PEIG does not exist in AIM → management - either it is an attack, or repair after AIM failure - reinit
UUS	Unknown user for secondary provider	Operation status or security problem - the identity does not exist in a secondary AIM → management - it is either the operation status, attack or repair after failure of AIM - init, or reinit at secondary AIM
UI	Invalid identification	Operation status - PEIG Operation status - PEIG or AIM determined that CyberID is violating terms of validity (time or number of uses) → management - e.g. rechange menu or change in access rights

Code	Brief description	Note
UPR	Unsupported security profile	AIM or PEIG cannot find (accept) the requested security profile → change of profile or denial of PEIG
UIP	Unsupported IL security profile	AIM or PEIG cannot find (accept) the requested IL (identity link) security profile → change in IL profile or request denial
UOP	Unsupported extension object profile	PEIG cannot accept the required security profile of the extension object → change of profile or request denial
VI	Valid identity	Functional problem or security incident - CyberID is valid upon requirement for rechange → probable use of rechange
NEO	Non-existing extension object	Functional problem - request for non-existing object → probable application error
NTD	Nothing to do	Normal - nothing to apply the request to → normal behaviour - failure (e.g. reading an empty list of objects) - application error or normal operation status
SPE	Error of second PEIG	Security of functional problem - attack or operation status or user error
UTL	Unsupported transition between security levels	Cannot accept the required change in security profile → change of request or acceptance of unknown CyberID
DLN	Requested login name is already used by another PEIG	Functional problem - duplicity of legacy login name → probable error of application or application user
NER	Insufficient rights to process the request	Functional problem → application error or user problem
DR	Duplicate replica	Functional problem → application error or user problem
NS	Non-existing session	Functional problem → application error or user problem
CTO	Exceeded max. communication time	Functional problem → configuration error or operation problem
ERR	Unspecified error	Security of functional problem: attack or implementation error, standard, general error message instead of specific ones
UV	Unsupported version	Functional problem (incompatibility) → management - change of request
DI	Repeated initialization	Functional problem or security incident - PEIG identified an attempt at repeated origin of a CyberID for the same AIM → either an error in the application that did not check the existence of the CyberID, or a security attack or configuration error - another AIM exists with the same SPID
CR	Applying a rejected change	Recovery from problem - consequence
MI	Missing identity	Functional problem or security incident - CyberID does not exist upon request for reinit → probable incorrect use of reinit
IE	Self-termination	PEIG security compromised→ management of the compromise, e.g. flagging an attribute and blocking access rights
NAP	Not accepted by PEIG	Secondary error - use primary
UCC	Incompatible keys	Functional problem - problem with keys in extension object → probable application error
NOP	No operation requested	Functional problem or attack → probable application error

Code	Brief description	Note
UIL	Unknown ILID	Functional problem or attack → probable application error
ILM	Missing ILID	Functional problem or attack → probable application error
ISE	Identity Link electronic stamp error	Functional problem or attack → probable application error
NSA	Missing address of secondary AIM	Functional problem or attack → probable application error
NU	Not unique	Internal alarm or functional or security problem
LI	Locked identity	Normal - identity removed from use by administrator or automatically → normal authentication failure
DMR	Duplicated meeting room	Normal—meeting room name conflict, the attempt to create a second meeting room with the same name results in a normal failure
UMR	Unknown meeting room	Normal—meeting room name conflict, the attempt to enter the non-existent meeting room results in a normal failure
CMR	Closed meeting room	Normal—the attempt to enter into a closed meeting room results in a normal failure
MET	Meeting room enter timeout	Normal—the second PEIG did not enter into the meeting room in time results in a normal failure
MCT	Meeting room confirmation timeout	Normal—the first PEIG did not confirm the second PEIG in time results in a normal failure
BIM	Binding item is missing	Security or functional problem—the required binding information is missing
BEE	Binding evaluation error	Security or functional problem—an attack to a banded channel or target application integration error
UBM	Unable binding mode	Security or functional problem—probably a target application integration error

Table 10-3: Error statuses of authentication process