Generic Open Link-Layer API for Unified Media Access—GOLLUM

D3.1
API Definition
(Final M24 version)

Contractual date of delivery to the CEC:	**31st August 2006**
Actual date of delivery to the CEC:	**31st August 2006**
Editor:	**Mahesh Sooriyabandara**
Authors:	**Mahesh Sooriyabandara, Tim Farnham, Costas Efthymiou (TREL), David Siorpaes (STM), Matthias Wellens, Janne Riihijärvi, Krisakorn Rerkrai (RWTH), Alain Gefflaut (EMIC), Arthur van Rooijen (MA)**
Participants:	**STM, RWTH, EMIC, UC, TID, TREL, MA**
Workpackage	**WP3**
Security:	**Public**
Nature:	**R**
Version:	**1.00**
Total number of pages:	**83**

Abstract:

This document specifies the Unified Link Layer API (ULLA) developed by the EC funded Gollum Project. The ULLA is an open, extensible, platform independent Application Programming Interface to access and control all types of wired and wireless link technologies (e.g. 802.11x, Bluetooth, GPRS/UMTS and ZigBee) in a uniform manner. The ULLA specification includes two APIs; Link User (LU) API and Link Provider (LP) API and a set of mandatory classes that abstracts the link technologies. The LU API is intended for application developers who are willing to use ULLA to control communication links available to them on a device and the LP API is intended for device manufacturers who wish to provide a ULLA compatible software layer over their proprietary device drivers.

Keyword list: Unified Link Layer API, Link User interface, Link Provider interface, API definition, ULLA API.

Document Revision History

Version	**Date**	**Author**	**Summary of main changes**
0.01	04/05/06	RWTH	Proposed table of contents.
0.05	12/06/06	TREL	Contributions to section 1 , 2 and Appendix A
0.08	12/06/06	STM	Contributions to section 4 and Appendix B
0.10	16/06/06	TREL	Updated text in section 1 and 2
0.20	28/06/06	RWTH	Contributions for sections 3 and 5.
0.25	04/07/06	RWTH	Updated section 3, 4 and 5 to reflect the latest agreed changes.
0.3	11/07/06	EMIC	Contributions for Appendix C
0.35	12/07/06	STM	Updated section 4
0.4	12/07/06	TREL	Edited & sections were formatted of the whole document
0.41	01/08/06	TREL	Updated the appendices
0.42	09/08/06	RWTH	Updated the document reflecting the latest smaller changes agreed between partners
0.50	18/08/06	TREL	Added “cancel command” function to LP If to make it consistent with the LU If. Some other minor changes.
1.00	30/08/06	RWTH	Finalized document

Contents

1. Introduction

This document specifies the Application Programming Interface (API) and related data definitions of Unified Link Layer API (ULLA). The ULLA is an open, extensible, platform independent API developed by the European Commission funded Gollum Project to enable access and control all types of wired and wireless link technologies in a technology-agnostic way. It is especially aimed at wireless mobile technologies, which include, but not limited to 802.11x, Bluetooth, GPRS/UMTS and ZigBee. This is an enabling technology aimed at different wireless stakeholders to promote advance and intelligent use of communication resources on different computing platforms such as laptops, PDAs, smart-phones and sensor devices. The primary audience for this document is two folds: application programmers who wish to use ULLA to control the communications links available on a device and device manufacturers who wish to provide a ULLA compatible software layer over their proprietary device drivers.

1.1. References

The following URLs and other pointers to various specifications and other documents are pertinent in understanding this specification. These can all be found at http://www.ist-gollum.org/deliverables.

D2.1 State-of-the-Art

> “This document provides a state of the art review on existing wireless technologies and associated APIs. It also presents some of the work being carried out by various industrial and academic projects related to the concept of Universal Link Layer API”.

D2.4 Final Architecture and API

> “This provides a complete description of the ULLA architecture, describing the use cases, interfaces, classes and various functional modules. The architecture is described using the UML diagrams”

D3.3 Validation Performance Report

> “This presents the results from the validation tests performed on the first public release of the ULLA API. It also presents results from the performance evaluation of a number of different ULLA implementations tested on different Operating Systems (OSs) and hardware platforms. This report also documents the details of the testing methods, test cases used”

D3.4 API Guidebook

> “This could be considered as a comprehensive implementation guide of the ULLA which acts as a programming reference for the programmers of ULLA cores, LUs and LLAs. It provides in-depth design details, recommendations and guidelines about implementation options for different operating systems and hardware platforms, and details on the advance use of ULLA”

1.2. Overview of the API

The ULLA provides a set of well defined functions to be used by Link Users to access information and issue commands to Link Providers. The ULLA specification describes the interfaces to LUs and LPs and core functionality. The API described in here presents the ULLA at two levels:

- **Link User API:** This is intended for application developers, who are willing to control wireless adapters as a whole with a high level of abstraction (Section 3).
- **Link Provider API:** This will be used by the device manufacturers to provide a ULLA compatible software layer (as an LLA – Link Layer Adaptor) over their technology and OS specific device drivers (Section 4).

1.3. Definitions

Link

A "Link" is a communications facility or medium over which network nodes can communicate at the Link Layer. Each Link is associated with a minimum of two endpoints. Multiple logical links can be supported over the same device with different properties (such as QoS or security settings). Links can support either connection oriented or connectionless communication modes. LLAs can register Links for pre-registered Link Providers to ULLA. For ULLA, a Link is exhaustively defined by a ullaLink class description.

With respect to the ISO/OSI or TCP/IP mode,l a Link is intended to be an interconnecting communication channel at layer-2. A pair of layer-2 addresses, which usually identify the two respective peers, uniquely identify each link. In this context, peers are the terminal installations, which implement the layer-2 communication protocol. For instance, a single network interface card (NIC) might be able to maintain multiple links with different peers.

Link User (LU)

Applications (user or system applications), communication middleware, transport entities or any other entity that register with ULLA as a Link User.

Here, the term application is understood in a software architectural sense. It does not limit the potential Link Users to "layer-7" applications with reference to a communication model, but also includes communication middleware, transport entities or routing agents.

Link Provider (LP)

Radio devices or other communication devices (or more specifically the driver software or other agent software associated with the devices) providing Links to be used by Link Users and selected and configured through the ULLA. A Link Layer Adapter (LLA) is a ULLA compatible software component used with legacy drivers in order to support the ULLA Link Provider interface. Conceptually, they act as a proxy agent for the real driver and adapt the command and event interactions accordingly. LLAs can register Link Providers with ULLA, defined by their Link Provider class description. A Link Provider itself hosts Links with the same ullaLink class description.

A Link Provider is an abstraction of a network interface card (NIC). For instance a LLA for a multi-mode NIC supporting GPRS and WLAN might register two Link Providers with ULLA. Each of them is defined by the means of a specific class description.

Commands

Commands are requests from Link Users (such as application or transport layer entities) sent to ULLA Core for configuring and controlling Link Providers. Commands may be passed to the Links or Link Providers in order to set specific parameters or instruct it to perform a specific operation at Link Provider level.

Most relevant commands on LPs include the possibility of discovering new Links. Links generally might not have so many commands, but for instance connect and disconnect are mandatory. Connectionless Links might not do anything there except returning OK.

Query

The primary mechanism used by Link Users for retrieving information using certain selection criteria. Queries are specified in a standard format using UQL (ULLA Query Language) which is a subset of SQL (Structured Query Language). A query could be used by Link Users to access information synchronously or to specify criteria for filtering and sending of notifications to Link Users asynchronously.

A query specifies a list of interesting attributes for the LU, e.g. bandwidth, and optionally a condition that all in the result listed LPs or Links have to meet. The scope of a query can be scaled down to a particular class, such as Cellular-Link-class or IEEE 802.11-LinkProvider-class. In sub-classes, more attributes are available, for instance cellular-type of link specific attributes. If an attribute is requested from a class which does not support it, the query fails. A freshness parameter indicates the requirements regarding the age of the addressed attributes.

Events

Events are passed from the LP or Link to the UEP (ULLA Event Processing). If the ULLA Core has registered its interest in the update of a particular attribute using the requestUpdate() function, the LP or Link reports either periodically or upon disposal of new measurement values.

Some attributes have to be updated by the LP or Link based on asynchronous events. For instance the Frame Error Rate could be updated when receiving a new frame. Other attributes are evaluated at periodic times, such as the current throughput (bytes within last period).

Link class description

A Link class description exhaustively characterizes the type of Link with a set of Link properties, also called attributes, and the available commands that can be issued on that type of Link.

Some interesting properties of a Link are connection oriented (infrastructure) versus connectionless (broadcast or ad-hoc) mode, security mode, and link quality parameters. A common command for Links is connect().

Link Layer Adapter (LLA)

Link Layer Adapter is a shim software layer between ULLA Core and a legacy Link provider, enabling Link Users to communicate with legacy LPs transparently through ULLA Core.

A LLA is a piece of software, which registers Link Providers and links to ULLA Core by the means of the defined interface. A LLA is typically tailored for a specific device driver or device driver tool.

Link Provider class description

A Link Provider class description characterizes the type of Link Provider with a set of Link Provider properties, also called attributes, and the available commands that can be issued on that type of Link Provider.

For instance Link Provider properties can capture power modes of the respective NIC. Commands may set the respective NIC mode in operation, which prevents communication on the hosted links, such as the scanning command might do.

Notification

A notification is sent from ULLA to the LU when a pre-registered UQL condition on LP or Link attributes is met. Additionally, they can indicate a change in LP or Link state, such as a new LP registered or Link deregistered. The UQL language is also used at registration time to specify the associated information coming alongside the notification.

This ULLA feature is very powerful for LUs that are interested in Link adaptation. Most applications might reside on highly portable attribute conditions for many Links, such as browsers, while others can make use of the full flexibility of UQL, such as connection managers or multimedia clients.

Sub-link

If a Link can be divided into a set of Links with the same layer-2 addresses, each instance is referred to as a sub-link.

For instance, sub-links might capture the ability for the link to support different service classes (such as different Bluetooth service profiles).

ULLA Core (UC)

ULLA Core is the module that implements all the functionalities of ULLA including the management of the interfaces towards Link Users and Link Providers. An ULLA Core is comprised of three functional components that deal with Command Processing, Query Processing and Event Processing.

ULLA Storage

ULLA keeps persistent information in the ULLA storage. This can rely on a legacy database or can be a custom realization tuned for a given platform.

The ULLA Storage system will typically contain link related information and parameters which will be accessed by the Link User by means of the ULLA Query Language. If necessary, ULLA queries can be translated into an implementation specific query language by the ULLA Core.

Unified Link Layer API (ULLA)

ULLA is an abstraction of an operating-system independent implementation of the defined standard interfaces, which Link Users and LLAs can use. Generally, the word ULLA can refer to the API as well as the system that realises the API.

2. Architecture

The aim of ULLA is to provide an API for accessing link-layer functionality and information in a technology independent manner. It hides network standards heterogeneity behind a standard set of functionalities applicable to a wide range of underlying link technologies. ULLA provides an abstraction from specific link technologies to the Link Users by regarding a link to be a generic means of providing a communications service. In this context, links are made available and configured through Link Providers, to permit abstraction from specific platforms and technologies. Link Users that benefit from ULLA services include, but are not limited to, any higher layer protocols, middleware or application software. Figure 2-1 presents a graphical overview of the different components in the ULLA architecture.

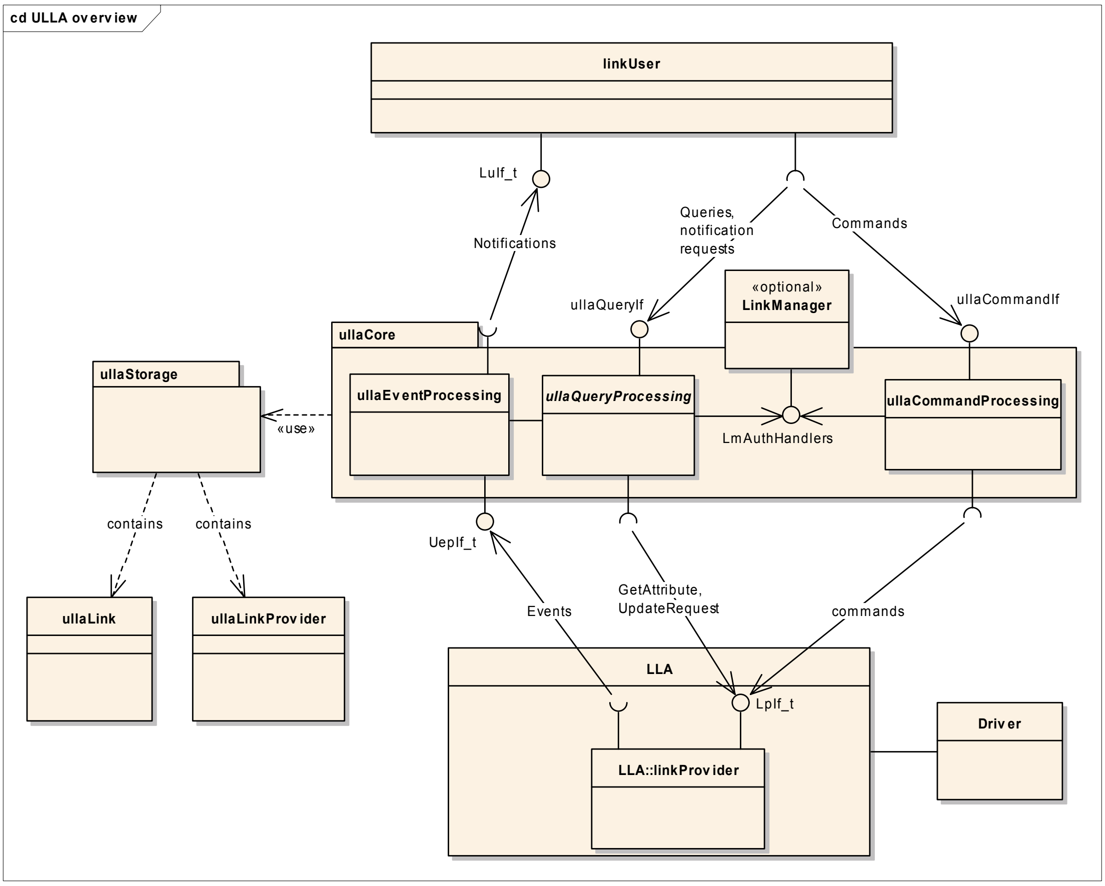

Figure 2-1: The ULLA Architecture.

2.1. *ULLA Core functionality*

ULLA not only provides a set of unique and well defined functions used by the Link Users to access relevant information and issue commands to Link Providers but also specifies core functionality (namely Command Processing, Event Processing and Query Processing) that enables these different services to be performed to aid the Link User.

In the centre of the picture, the ULLA Core is represented; which contains three main functional components. The *UllaQueryProcessing* is in charge of analyzing the queries and notification requests coming from Link Users. The *UllaCommandProcessing* handles commands and forwards them to the corresponding Link Provider. The *UllaEventProcessing* module takes care of handling events arriving from the Link Providers (new Link arrivals, new value for a Link attribute, etc..) and in particular the evaluation of registered notification requests. Finally, the *Ulla Storage*, represented outside of the Ulla Core, is a component used to cache link attributes collected from the Links and Link Providers in order to avoid access to the drivers or hardware for each query. This storage facility also holds both static and dynamic information regarding all ULLA entities such as Link Provider and Link User characteristics, historical statistics, channel information, mutual exclusion characteristics, and others.

The main services provided through the ULLA are:

- **Queries**: A generic querying mechanism allowing applications to retrieve link information in a technologically independent fashion. To enable this, a query language called ULLA Query Language, a subset of the well-known SQL, is used.
- **Commands**: A mechanism that allows applications to configure and manage Links in a standard way using commands that can be called from user level applications or from lower level entities such as Connection Managers, Link Managers or other middleware.
- **Events**: An asynchronous notifications mechanism based on user defined link criteria. Link Users can define and register any type of conditions triggering an asynchronous notification through UQL. Once registered, these conditions are dynamically evaluated by the ULLA Core based on events reported by the Link Providers. For example, it is trivial with ULLA to enable a notification when received signal strength goes under a certain threshold of a specific link.

In addition to these, ULLA also supports a concept called role management to handle LUs with different privileges. LU's ability to use a set of services provided by ULLA is controlled based on the role and privileges associated with it. This adds a layer of security to ULLA to prevent undesired behaviour (in terms of link connection/disconnection, bad configurations, etc.) that could possibly disrupt the user experience, behaviour detrimental to overall system efficiency, etc. Role management also enables the prioritisation of Link Users so that conflicts between them can be easily resolved by a Link Manager or other arbitration entity.

2.2. ULLA Interfaces

In order to achieve the above objectives, the ULLA API is separated into varaious parts to accommodate the distinction between different software domains, which are uncorrelated with each other. The first part of the API, the ULLA Link User API, is intended for application developers who are willing to control wireless adapters as a whole with a very high level of abstraction. The second part, the ULLA Link Provider API, will be used by device manufacturers who are more interested in providing a ULLA compatible software layer (as an LLA) over their technology and OS specific device drivers. In the LLA context, the API definition is more tightly coupled with the concept of an LP mapped to a physical network adapter.

2.3. Link User Interface Usage

In order for a LU to use ULLA, it needs to register itself with the ULLA Core. The function *ullaRegisterLu()* is used to pass a *LuDescr_t* structure containing the name, description and apiVersion and LU role to the ULLA Core. The role specifies the type of access or services the LU expects to receive from the ULLA Core. The "ULLA role model" described in the current ULLA specification groups one or more ULLA services (i.e. function calls) into different role categories. For example, the basic role, namely the ULLA standard Link User role (i.e. ULLA_ROLE_STD_LU), will only provide the query interface to the LU, which allows for read access to a limited number of ULLA objects. It does not provide any command capability or write (i.e. access using *ullaSetAttribute()*) access. Details of each role and their corresponding service bundles (or allowed function calls) are described in Appendix B.1.6.

A LU must call the *ullaUnregisterLu()* method when it stops using ULLA functionality.

2.3.1. Command Handling

The LU has the ability to send commands to the LPs and Links via the ULLA Core. The *ullaDoCmd()* method is used to execute a synchronous command. The LU needs to pass a *cmdDescr_t* command description structure; this structure contains the name of the command to be executed. It also passes the ID of the LP or Link and the name of the class that has to execute the command.

The *ullaRequestCmd()* method is used to execute an asynchronous command. In addition to the previous parameters, there is also a need to pass a pointer to a callback routine of type *handleAsyncCmd_t*. This handles the result of an asynchronous command. The ullaRequestCmd() method will return a unique identifier (cmdId) when the command is queued. The LU can cancel a queued command by using the *ullaCancelCmd* method by passing *cmdId* as a parameter.

The *ullaPrepareCmd()* method is used by the LU to request a lock for a single command to be executed. The lock is only valid for a certain time. This enables the integrity of a single command by making sure that only one LU is able to configure an entity at a given instance.

In case several LUs issue conflicting commands to the same link, an arbiter called Link Manager can apply suitable policies to resolve conflicts. An interface is defined in the current design to ease the insertion of a third-party LM in the ULLA software architecture.

2.3.2. Query Processing

The LU can retrieve information from the ULLA Core or LPs by doing an information request. This is done with the *ullaRqustInfo()* method; with this method, the LU passes a query string and a pointer to an *ullaResult_t* type handle. The data in the result handle can be retrieved using special *ullaResult* accessor functions. Since memory allocation for the result set is done by the ULLA Core, after all the data has been retrieved, the LU needs to call the *ullaResultFree()* method to free the memory.

Sometimes it is necessary to be informed when a certain attribute of a Link is changed. This can be done by requesting a notification. This works in a similar way to the *ullaRequestCmd()* method. The *ullaRequestNotification()* method is used for this; a *RnDescr_t* structure and a pointer to a callback function are passed as parameters. The function will return an *rnId* when the notification has been successfully queued. This notification request can be cancelled by the *ullaCancelNotification()* method.

2.4. Link Provider Interface

This interface provides the means to notify link events received from the LP to ULLA Core and to pass commands received from LUs to LPs. When registering using the *registerLp()* method, a LP needs to pass the structure *LpDescr_t*. This structure contains the version of the ULLA API the Link provider is compatible with and a pointer to its interface. Upon successful registration, the ULLA Core returns a unique identifier (*lpId*). LP registration does not automatically imply that there are also active links. The ULLA Core needs to do an active scan to find all available links. When an LP is unloaded it must call the *unregisterLp()* method. It needs to pass the *lpId* it received when registering.

2.4.1. Command Handling

Commands are passed (through the ULLA Core) from the LU to the LP. The *execCmd* method is used to pass a synchronous command to the LP along with identification of the Link or Link provider that has to execute the command. The ULLA Core uses the *lpId* in the *cmdDescr_t* to address the appropriate LP. The *cmdDescr* contains all other information the LP needs. In the current API definition, the LP has no special function to pass an asynchronous command. The handling of asynchronous commands is completely done by the ULLA Core. From the LPs point of view, the *execCmd* method is used in both cases. However, the command can be cancelled by invocation of the *cancelCmd* function.

2.4.2. Query Handling

The ULLA Core will normally first retrieve data from the ULLA Storage. When newer information than that which is stored is needed, the ULLA Core can retrieve this information by using the *getAttribute* method in order to fetch the current value of an attribute from a LP or Link. The ULLA Core needs to pass *AttrDescr_t* data structure with id of Link or LP, class name, the attribute name, a data qualifier and a pointer to where the result has to be stored as paramter. With the data qualifier, the LP can tell if the value is a HARDCODED, THEORETICAL, ESTIMATED, MEASURED or EXACT value. The memory to store the value is allocated by the LP. This memory then has to be freed by the ULLA Core with the *freeAttribute* method when it is no longer required.

When the LU requests a notification, e.g. when a certain attribute has changed, the ULLA Core will call *requestUpdate*. As parameters, the ULLA Core needs to supply a structure with requested attribute and a *RuDescr_t* structure, with information regarding the count and time interval the update needs to be sent. When the attribute is updated the LP will call the handleEvent function of the ULLA Core Event Processing Interface. The update request can be cancelled with the *cancelUpdate* method. In order to track all the different update requests sent to the LPs, ULLA Core will generate a unique ID for all requests.

3. Link User Interface

The Link User Interface forms the upper part of the Unified Link-Layer API, and it is offered by the ULLA Core to Link Users. In the following, the details about the calls used for each of the offered and the related definitions are first presented. Then, the ULLA Query Language is introduced, which is used in several of the LU interface functions to specify information requests in a standard manner.

This section will introduce all functions that are offered by ULLA Core, as part of the Link User Interface, grouped according to their functionality to distinguish different ULLA features available to application programmers.

3.1. ULLA General Functions

The ULLA Query Processing is part of ULLA Core and offers query and notifications features to the LU.

3.1.1. ullaGetCoreDescriptor()

SYNTAX

```
ULLA_API ullaResultCode ullaGetCoreDescriptor (INOUT CoreDescr_t
    *coreDescriptor)
```

DESCRIPTION

Get information about the ULLA Core version.

ARGUMENTS

The parameter `coreDescriptor` is the structure containing information about ULLA manufacturer, version and supported profiles. The structure memory is allocated by the ULLA client and the Core implementation will fill it in.

RETURNS

The `ullaGetCoreDescriptor()` function will return `ULLA_OK` if the function is successfully completed and it will return `ULLA_ERROR_INVALID_PARAMETER` if the parameter is wrong.

3.1.2. ullaRegisterLm()

SYNTAX

```
ULLA_API ullaResultCode ullaRegisterLm (IN LuDescr_t *luDescr, IN
    LmAuthorizationHandlers_t *auth)
```

DESCRIPTION

The function must be called by the Link Manager to register itself with the ULLA Core at init time. Since external LM is optional, if an ULLA Core does not support it, it must return `ULLA_ERROR_LM_NOT_SUPPORTED`.

ARGUMENTS

The parameter `luDescr` is the structure containing information about the Link Manager. The parameter `auth` is the structure that includes handlers to authorization handling functions that are implemented by the Link Manager.

RETURNS

The function will return `ULLA_OK` if the registration is successful, it will return `ULLA_ERROR_LM_NOT_SUPPORTED` if the ULLA Core does not support external LM, it will return `ULLA_ERROR_API_VERSION_MISMATCH` if the ULLA version requested by the LU does not match with the ULLA Core version, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong.

3.1.3. ullaRegisterLu()

SYNTAX

```
ULLA_API ullaResultCode ullaRegisterLu (IN LuDescr_t *luDescr, IN
    LuRole_t luRole)
```

DESCRIPTION

The `ullaRegisterLu()` function must be called by every LU wishing to use ULLA Core functionalities. By calling this function, an LU gets registered within the ULLA Core.

ARGUMENTS

The `luDescr` is the structure containing information on the LU and the `luRole` is the identifier of the Link User Role that the LU wants to register.

RETURNS

The function will return `ULLA_OK` if the registration is successful, it will return `ULLA_ERROR_API_VERSION_MISMATCH` if the ULLA version requested by the LU is not compatible with the available ULLA API. The function will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong, it will return `ULLA_ERROR_UNSUPPORTED_ROLE` if the requested role is not supported, it will return `ULLA_ERROR_ROLE_DENIED` if the ULLA core denies the requested role, it will return `ULLA_ERROR_UNSUPPORTED_PROFILE` if the requested profile type is not supported by the ULLA Core, and it will return `ULLA_ERROR_ALREADY_REGISTERED` if the LU is already registered.

3.1.4. ullaUnregisterLu()

SYNTAX

```
ULLA_API ullaResultCode ullaUnregisterLu()
```

DESCRIPTION

The `ullaUnregisterLu()` function should be called by LUs when stopping using ULLA functionality, e.g. upon termination. When this method is called, the LU entry within the ULLA Storage is removed, and all pending notifications are canceled.

ARGUMENTS

None.

RETURNS

The function returns `ULLA_OK` if the operation is successful and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.1.5. ullaSetAttribute()

SYNTAX

```
ULLA_API ullaResultCode ullaSetAttribute (IN AttrDescr_t *attrDescr)
```

DESCRIPTION

Sets a writeable Link or Link Provider attribute.

ARGUMENTS

The `attrDescr` structure contains the details of the attribute that needs to be set. It also includes the class the attribute is part of.

RETURNS

The function will return `ULLA_OK` if the attribute is successfully updated, it will return `ULLA_ERROR_SETATTR_NOT_ALLOWED` if the LU is not allowed to set the attribute, and it will return `ULLA_ERROR_ALREADY_LOCKED` if there is already a lock on the Link or LP, it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet, it will return `ULLA_ERROR_SETATTR_READONLY` if the attribute to be set is read only, it will return `ULLA_ERROR_INVALID_ATTRIBUTE` if an invalid attribute name was provided in the `attrDescr`, it will return `ULLA_ERROR_INVALID_CLASS` if the class name provided in the `attrDescr` is invalid, it will return `ULLA_ERROR_UNKNOWN_ID` if the Link or LP identifier provided in the `attrDescr` is not known, it will return `ULLA_ERROR_INVALID_VALUE` if the value to be set is invalid, it will return `ULLA_ERROR_INVALID_QUALIFIER` if the qualifier to bet set is invalid, and it will return `ULLA_ERROR_SETATTR_NOTMULTIPLE` if several attributes were provided although a single value attribute is to be set.

3.2. ULLA Query Processing

3.2.1. ullaRequestInfo()

SYNTAX

```
ULLA_API ullaResultCode ullaRequestInfo (IN ULLA_STRING_t query, OUT
    ullaResult_t *result, ULLA_INT_t validity)
```

DESCRIPTION

The `ullaRequestInfo()` function allows an application to query the ULLA Storage using UQL as specified in section 3.10. This function call is synchronous.

ARGUMENTS

The parameter `query` is the query string in UQL, the parameter `result` is the pointer to the identifier of the result set to be returned. The identifier must be declared by the LU in advance. The `parameter` validity is longest accepted newness of the returned data in ms.

RETURNS

The function will return `ULLA_OK` if successfully completed, it will return `ULLA_ERROR_INVALID_CLASS` if the class in the query is wrong, it will return

ULLA_ERROR_INVALID_ATTRIBUTE if the attribute in the query is wrong, it will return ULLA_ERROR_SYNTAX_ERROR if the UQL query contains a syntax error, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong, it will return ULLA_ERROR_QUERY_NOT_ALLOWED if the LU is not allowed to perform the query, and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.2.2. ullaRequestNotification()

SYNTAX

ULLA_API ullaResultCode ullaRequestNotification (IN RnDescr_t *rndescr, IN handleNotification_t handler, OUT RnId_t *rnId, ULLA_INT_t validity)

DESCRIPTION

The ullaRequestNotification() function allows a LU to request event-driven or periodic notifications from the ULLA Core. A ullaRequestNotification() shall remain in force until specifically cancelled by invoking the function ullaCancelNotification(). This function is asynchronous.

ARGUMENTS

The rndescr is an RnDescr_t structure containing details on the type of notification which is being requested, the handler is the callback function provided by the application to be called by ULLA Core when reporting events. The rnId is the request notification identifier returned by ULLA Core. The parameter validity is longest accepted newness of the returned data in ms.

RETURNS

The function will return ULLA_OK if successfully completed, it will return ULLA_ERROR_INVALID_CLASS if the class in the query is not correct, it will return ULLA_ERROR_INVALID_ATTRIBUTE if the attribute in the query is not correct, it will return ULLA_ERROR_SYNTAX_ERROR if a syntax error is found in the UQL syntax, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong, it will return ULLA_ERROR_QUERY_NOT_ALLOWED if the LU is not allowed to perform the query, it will return ULLA_ERROR_PERIOD_TOO_SHORT if the notification period is to short, so that ULLA Core cannot handle it, it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet, and it will return ULLA_ERROR_INVALID_HANDLER if the pointer to the handler is illegal.

3.2.3. ullaCancelNotification()

SYNTAX

ULLA_API ullaResultCode ullaCancelNotification (IN RnId_t rnId)

DESCRIPTION

The ullaCancelNotification() function allows a LU to cancel a notification previously requested with ullaRequestNotification().

ARGUMENTS

The parameter rnId is the numeric identifier of the notification to be canceled, as returned by ullaRequestNotification().

RETURNS

The function will return ULLA_OK if the cancellation is successfully completed, it will return ULLA_ERROR_INVALID_NOTIFICATION if the notification with the requested rnId does not exist or has already been canceled, and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.2.4. ullaResultFree()

SYNTAX

```
ULLA_API ullaResultCode ullaResultFree (IN ullaResult_t res)
```

DESCRIPTION

In order to free the memory allocated by ullaRequestInfo() or ullaRequestNotification() for the result set, a LU has to call the ullaResultFree() function after a given result set has been processed.

ARGUMENTS

The parameter res is the identifier of the result set which is to be deallocated.

RETURNS

The function return ULLA_OK if successfully completed, it will return ULLA_ERROR_INVALID_ULLARESULT if the result set does not exist or has been deallocated already, and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.2.5. handleNotification_t()

SYNTAX

```
typedef void(*handleNotification_t)(IN RnId_t rnId, IN ullaResult_t
    res, void *privdata)
```

DESCRIPTION

This is the type of callback function that must be provided by the LU when calling ullaRequestNotification().

ARGUMENTS

The rnId is the numeric identifier, as returned by ullaRequestNotification(), of the notification request that triggered the notification event. The parameter res is the identifier of the result set returned by the notification. It is supposed that, when calling ullaRequestNotification(), the LU specifies which attributes should be returned with the notification. These parameters are returned by the ULLA Core in a result set of type ullaResult_t, i.e. the type used for results returned by ullaRequestInfo(). As a consequence, result data must be accessed using the appropriate accessor functions (ullaResultNextTuple(), ullaResultIntValue(), etc.). This result also needs to be freed by the Link User with the ullaResultFree() function. The final parameter privdata is the LU private data, which is passed upon calling ullaRequestNotification().

3.3. ULLA Accessor Functions

The accessor functions are used to access the data in an ullaResult_t result set. Some of the functions return the length of a string. Here the same rules apply as in normal C runtime functions. The length returned is always the length of the string without the '\0' terminator. Important is that, when retrieving the string, one byte is added to the length to store this '\0' terminator.

3.3.1. ullaResultNumFields()

SYNTAX

```
ULLA_API ullaResultCode ullaResultNumFields (IN ullaResult_t res, OUT
     ULLA_INT_t *num)
```

DESCRIPTION

The `ullaResultNumFields()` gives information on how many fields (e.g. columns) are contained within a result set.

ARGUMENTS

The parameter `res` is the identifier of the result set to be analyzed, as returned by `ullaRequestInfo()`, the `num` is the number of fields.

RETURNS

The function will return `ULLA_OK` if successfully completed, it will return `ULLA_ERROR_INVALID_ULLARESULT` if the result identifier `res` is invalid, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.3.2. ullaResultNumTuples()

SYNTAX

```
ULLA_API ullaResultCode ullaResultNumTuples (IN ullaResult_t res, OUT
     ULLA_INT_t *num)
```

DESCRIPTION

The `ullaResultNumTuples()` function gives information on how many rows (e.g. tuples) are contained within a result set.

ARGUMENTS

The parameter `res` is the identifier of the result set to be analyzed, as returned by `ullaRequestInfo()`, the second parameter `num` provides the number of tuples.

RETURNS

The function will return `ULLA_OK` if successfully completed, it will return `ULLA_ERROR_INVALID_ULLARESULT` if the result identifier `res` is invalid, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.3.3. ullaResultFieldName()

SYNTAX

```
ULLA_API ullaResultCode ullaResultFieldName (IN ullaResult_t res, IN
    ULLA_INT_t fieldNo, OUT ULLA_STRING_t name, INOUT ULLA_INT_t
    *size)
```

DESCRIPTION

The ullaResultFieldName() function returns the name of a field in a result set given its column number.

ARGUMENTS

The parameter res is the identifier of the result set to be analyzed, as returned by ullaRequestInfo() and the parameter fieldNo is the column number of the field of which the name is to be retrieved. The column numbers start from 1. The parameter name is the buffer where to store the field name, allocated by the LU. The last parameter size is the length of the buffer allocated. The length of the result string is returned (without '\0' terminator). If size is set to 0 when calling the function the length needed for the string without '\0' terminator is returned.

RETURNS

The function will return ULLA_OK if successfully completed, it will return ULLA_ERROR_INVALID_ULLARESULT if the result set does not exist or has been deallocated, it will return ULLA_ERROR_INVALID_FIELD if the requested field does not exist, it will return ULLA_ERROR_BUFFER_TOO_SMALL if the buffer size is insufficient, it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet, and it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong.

3.3.4. ullaResultFieldNumber()

SYNTAX

```
ULLA_API ullaResultCode ullaResultFieldNumber (IN ullaResult_t res,
    IN ULLA_STRING_t fieldName, OUT ULLA_INT_t *num)
```

DESCRIPTION

The ullaResultFieldNumber() function returns the number of a field in a result set given its name.

ARGUMENTS

The parameter res is the identifier of the result set to be analyzed, as returned by ullaRequestInfo(). The parameter fieldName is the name of the field of which the column number is to be retrieved. The parameter num is the column number of the field. Column numbers start from 1.

RETURNS

The function will return ULLA_OK if successfully completed, it will return ULLA_ERROR_INVALID_ULLARESULT if the result set does not exist or has been deallocated, it will return ULLA_ERROR_INVALID_FIELD if the requested field does not exist, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.3.5. ullaResultValueLength()

SYNTAX

```
ULLA_API ullaResultCode ullaResultValueLength (IN ullaResult_t res,
    IN ULLA_INT_t fieldNo, OUT ULLA_INT_t *size)
```

DESCRIPTION

The `ullaResultValueLength()` function returns the length (in bytes) of a field within the current row of a specified result set. When the field contains a string the number of characters without the trailing '\0' terminator will be returned. If a field contains multiple strings the length of the longest string will be returned.

ARGUMENTS

The parameter `res` is the identifier of the result set to be analyzed, as returned by `ullaRequestInfo()`, the `fieldNo` is the column number of the field of which the length is to be retrieved. Column numbers start from 1. The parameter `size` gives the requested size n in bytes.

RETURNS

The function will return `ULLA_OK` if successfully completed, it will return `ULLA_ERROR_INVALID_ULLARESULT` if the result set does not exist or has been deallocated, it will return `ULLA_ERROR_INVALID_FIELD` if the requested field does not exist, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong, it will return `ULLA_ERROR_NO_CURRENT_TUPLE` if `ullaResultNextTuple()` has not been called, yet, or has been called called after `ullaResultNextTuple()` returned `ULLA_ERROR_NO_MORE_TUPLES`, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.3.6. ullaResultNumFieldValues()

SYNTAX

```
ULLA_API ullaResultCode ullaResultNumFieldValues (IN ullaResult_t
    res, IN ULLA_INT_t fieldNo, OUT ULLA_INT_t *num)
```

DESCRIPTION

This function returns the number of values a given field is composed of. The field within the current row is evaluated. The same field is allowed to have different number of values in different rows within the current result set.

ARGUMENTS

The parameter `res` is the identifier of the result set to be analyzed, as returned by `ullaRequestInfo()`, the parameter `fieldNo` is the column number of the field of which the number of values is to be retrieved. Column numbers start from 1. The parameter `num` is the requested number of values.

RETURNS

The function will return `ULLA_OK` if successfully completed, it will return `ULLA_ERROR_INVALID_ULLARESULT` if the result set does not exist or has been deallocated, it will return `ULLA_ERROR_INVALID_FIELD` if the requested field does not exist, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong, it will return `ULLA_ERROR_NO_CURRENT_TUPLE` if `ullaResultNextTuple()` has not been called, yet, or has been called after `ullaResultNextTuple()` returned `ULLA_ERROR_NO_MORE_TUPLES`, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.3.7. ullaResultValueType()

SYNTAX

```
ULLA_API ullaResultCode ullaResultValueType (IN ullaResult_t res, IN
    ULLA_INT_t fieldNo, OUT BaseType_t *type)
```

DESCRIPTION

This function returns an integer code representing the type of all values contained within a given field. If a field contains multiple values these will all be of the same type.

ARGUMENTS

The parameter `res` is the identifier of the result set to be analyzed, as returned by `ullaRequestInfo()`, `fieldNo` is the column number of the field of which the type is to be retrieved. Column numbers start from 1. The parameter `type` is a `BaseType_t` code (see section B.1.1) representing the data type.

RETURNS

The function will return `ULLA_OK` if successfully completed, it will return `ULLA_ERROR_INVALID_ULLARESULT` if the result set does not exist or has been deallocated, it will return `ULLA_ERROR_INVALID_FIELD` if the requested field does not exist, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong, it will return `ULLA_ERROR_NO_CURRENT_TUPLE` if `ullaResultNextTuple()` has not been called, yet, or has been called after `ullaResultNextTuple()` returned `ULLA_ERROR_NO_MORE_TUPLES`, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.3.8. ullaResultNextTuple()

SYNTAX

```
ULLA_API ullaResultCode ullaResultNextTuple (IN ullaResult_t res)
```

DESCRIPTION

The `ullaResultNextTuple()` sets the current row of the result set with the given identifier to the next row. The ULLA core keeps track internally of the current row for each allocated result set. The function `ullaResultNextTuple()` must be called at least once before retrieving data, in order to check if the result set is empty and to set the current row to the first row.

ARGUMENTS

The parameter `res` is the identifier of the result set as returned by `ullaRequestInfo()`.

RETURNS

The function will return `ULLA_OK` if successfully completed, it will return `ULLA_ERROR_INVALID_ULLARESULT` if the result set does not exist or has been deallocated, it will return `ULLA_ERROR_NO_MORE_TUPLES` if there are no more tuples to retrieve, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.3.9. ullaResultStringValue()

SYNTAX

```
ULLA_API ullaResultCode ullaResultStringValue (IN ullaResult_t res,
    IN ULLA_INT_t fieldNo, OUT ULLA_STRING_t str, INOUT ULLA_INT_t
    *size)
```

DESCRIPTION

The `ullaResultStringValue()` returns a string containing the value of the field with the given column number from the current row in the given result set. The string is returned including the trailing '\0' terminator. If a field contains multiple values, subsequent calls to this function are needed in order to retrieve all values. When all values are retrieved the function will return `ULLA_ERROR_NO_MORE_VALUES`.

ARGUMENTS

The parameter `res` is the identifier of the result set from which to get the value, `fieldNo` is the column number of the field from which the value is to be retrieved. Column numbers start from 1. The parameter `str` is the pointer to the string buffer where to store the null-terminated string value, `size` is the length of the buffer allocated. The length of the result string is returned (without '\0' terminator). If `size` is set to 0 when calling the function the length needed for the string without '\0' terminator is returned.

Note: The length can also be retrieved with the `ullaResultFieldLength()` method.

RETURNS

The function will return `ULLA_OK` if successfully completed, it will return `ULLA_ERROR_INVALID_ULLARESULT` if the result set does not exist or has been deallocated, it will return `ULLA_ERROR_INVALID_FIELD` if the requested field does not exist, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong, it will return `ULLA_ERROR_NO_CURRENT_TUPLE` if `ullaResultNextTuple()` has not been called, yet, or has been called after `ullaResultNextTuple()` returned `ULLA_ERROR_NO_MORE_TUPLES`, it will return `ULLA_ERROR_BUFFER_TOO_SMALL` if the buffer size is insufficient, it will return `ULLA_ERROR_NO_MORE_VALUES` if the field contains no more values, it will return `ULLA_ERROR_TYPE_MISMATCH` if the field does not contain a string, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.3.10. ullaResultIntValue()

SYNTAX

```
ULLA_API ullaResultCode ullaResultIntValue (IN ullaResult_t res, IN
    ULLA_INT_t fieldNo, OUT ULLA_INT_t *value)
```

DESCRIPTION

The `ullaResultIntValue()` returns as an integer the value of the field with the given column number from the current row in the given result set. If a field contains multiple values, subsequent calls to this function are needed in order to retrieve all values. When all values are retrieved, the function will return `ULLA_ERROR_NO_MORE_VALUES`.

ARGUMENTS

The parameter `res` is the identifier of the result set from which to get the value, `fieldNo` is the column number of the field from which the value is to be retrieved. Column numbers start from 1. The parameter value is the pointer to the location where to store the integer value.

RETURNS

The function will return ULLA_OK if successfully completed, it will return ULLA_ERROR_INVALID_ULLARESULT if the result set does not exist or has been deallocated, it will return ULLA_ERROR_INVALID_FIELD if the requested field does not exist, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong, it will return ULLA_ERROR_NO_CURRENT_TUPLE if ullaResultNextTuple() has not been called, yet, or has been called after ullaResultNextTuple() returned ULLA_ERROR_NO_MORE_TUPLES, it will return ULLA_ERROR_NO_MORE_VALUES if the field contains no more values, it will return ULLA_ERROR_TYPE_MISMATCH if the field does not contain an integer, and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.3.11. ullaResultDoubleValue()

SYNTAX

```
ULLA_API ullaResultCode ullaResultDoubleValue (IN ullaResult_t res,
    IN ULLA_INT_t fieldNo, OUT ULLA_DOUBLE_t *value)
```

DESCRIPTION

The ullaResultDoubleValue() returns as a double the value of the field with the given column number from the current row in the given result set. If a field contains multiple values, subsequent calls to this function are needed in order to retrieve all values. When all values are retrieved, the function will return ULLA_ERROR_NO_MORE_VALUES.

ARGUMENTS

The parameter res is the identifier of the result set from which to get the value, fieldNo is the column number of the field from which the value is to be retrieved. Column numbers start from 1. The parameter value is the pointer to the location where to store the double value.

RETURNS

The function will return ULLA_OK if successfully completed, it will return ULLA_ERROR_INVALID_ULLARESULT if the result set does not exist or has been deallocated, it will return ULLA_ERROR_INVALID_FIELD if the requested field does not exist, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong, it will return ULLA_ERROR_NO_CURRENT_TUPLE if ullaResultNextTuple() has not been called, yet, or has been called after ullaResultNextTuple() returned ULLA_ERROR_NO_MORE_TUPLES, it will return ULLA_ERROR_NO_MORE_VALUES if the field contains no more values, it will return ULLA_ERROR_TYPE_MISMATCH if the field does not contain a double, and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.3.12. ullaResultRawDataValue()

SYNTAX

```
ULLA_API ullaResultCode ullaResultRawDataValue(IN ullaResult_t res,
    IN ULLA_INT_t fieldNo, OUT ULLA_RAWDATA_t buf, INOUT ULLA_INT_t
    *size);
```

DESCRIPTION

The ullaResultRawDataValue() returns the value of the field with the given column number as raw data (bytes) from the current row in the given result set. If a field contains multiple values, subsequent calls to this function are needed in order to retrieve all values.

ARGUMENTS

The parameter res is the identifier of the result set from which to get the value, fieldNo is the column number of the field from which the value is to be retrieved. Column numbers start from 1. The parameter str is the pointer to the buffer where to store the bytes. The parameter size is the length of the buffer allocated. If length is set to 0 when calling the function the size needed for the buffer will be returned.

RETURNS

The function will return ULLA_OK if successfully completed, it will return ULLA_ERROR_INVALID_ULLARESULT if the result set does not exist or has been deallocated, it will return ULLA_ERROR_INVALID_FIELD if the requested field does not exist, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong, it will return ULLA_ERROR_NO_CURRENT_TUPLE if ullaResultNextTuple() has not been called, yet, or has been called after ullaResultNextTuple() returned ULLA_ERROR_NO_MORE_TUPLES, it will return ULLA_ERROR_BUFFER_TOO_SMALL if the buffer size is insufficient, it will return ULLA_ERROR_NO_MORE_VALUES if the field contains no more values, it will return ULLA_ERROR_TYPE_MISMATCH if the field does not contain raw data, and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.3.13. ullaResultValueQualifier()

SYNTAX

```
ULLA_API ullaResultCode ullaResultValueQualifier (IN ullaResult_t
    res, IN ULLA_INT_t fieldNo, OUT AttrQual_t *qualifier)
```

DESCRIPTION

This function returns the qualifier of the specified field within the current row of the given result set.

When a field contains multiple values, this function will return the qualifier of the current value. Using this function will *not* increment the value pointer.

ARGUMENTS

The parameter res is the identifier of the result set under evaluation, fieldNo is the column number of the field from which the value is to be retrieved. Column numbers start from 1. The parameter qualifier is the pointer to the location where to store the qualifier.

RETURNS

The function will return ULLA_OK if successfully completed, it will return ULLA_ERROR_INVALID_ULLARESULT if the result set does not exist or has been deallocated, it will return ULLA_ERROR_INVALID_FIELD if the requested field does not exist, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong, and it will return ULLA_ERROR_NO_CURRENT_TUPLE if ullaResultNextTuple() has not been called, yet, or has been called after ullaResultNextTuple() returned ULLA_ERROR_NO_MORE_TUPLES, and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.4. ULLA Command Processing

The ULLA Command Processing enables the LU to issue certain commands on LPs.

3.4.1. ullaPrepareCmd()

SYNTAX

`ULLA_API ullaResultCode ullaPrepareCmd (IN CmdDescr_t *cmddescr)`

DESCRIPTION

The function `ullaPrepareCmd()` sets a lock on a Link before executing a command. This function allows an LU to request a lock for single command to be executed on a specific LP or Link. The command call is synchronous. The lock is only valid for a certain time or until completion of the associated `ullaDoCmd()` or `ullaRequestCmd()`.

ARGUMENTS

The parameter `cmddescr` is the command description.

RETURNS

The function will return `ULLA_OK` if the lock has been successfully set, it will return `ULLA_ERROR_INVALID_COMMAND` if the command cannot be executed by Link or LP, it will return `ULLA_ERROR_INVALID_CLASS` if there is an invalid classname in cmddescr, it will return `ULLA_ERROR_UNKNOWN_ID` if there is an invalid link/lp ID in cmddescr, it will return `ULLA_ERROR_CMD_NOT_ALLOWED` if the LU is not allowed to execute the command, it will return `ULLA_ERROR_ALREADY_LOCKED` if there is already a lock on the Link or LP, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.4.2. ullaDoCmd()

SYNTAX

```
ULLA_API ullaResultCode ullaDoCmd (IN CmdDescr_t * cmddescr, IN
    ULLA_INT_t timeoutValue)
```

DESCRIPTION

Send synchronous commands to single a Link or Link Provider through ULLA. This method allows a LU to request a single command to be executed on a specific Link/LP. The command call is synchronous, i.e. the `ullaDoCmd()` function returns only on command completion or upon timeout expiration.

ARGUMENTS

The `cmddescr` provides a description of the command and the `timeOutValue` defines the maximum time available for completing the command.

RETURNS

The function `ullaDoCmd()` returns `ULLA_OK` upon successful command execution. It will return `ULLA_ERROR_INVALID_COMMAND` if the command cannot be executed by the addressed link or LP, it will return `ULLA_ERROR_INVALID_CLASS` if there is an invalid classname in cmddescr, it will return `ULLA_ERROR_UNKNOWN_ID` if there is an invalid link/lp ID in cmddescr, it will return `ULLA_ERROR_CMD_NOT_ALLOWED` if the user does not have sufficient privileges to execute the command. The return value will be

ULLA_ERROR_ALREADY_LOCKED if there is already a lock on the Link or LP, it will be ULLA_ERROR_TIMEOUT if the execution of the command has timed out, the function will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet, and it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong. Or if one of the required attribute values required to perform the command is not set properly then the error ULLA_ERROR_ATTRIBUTE_VALUE_INVALID will be returned. If the command cannot be completed for another reason the ULLA_ERROR_COMMAND_FAILED code is returned.

3.4.3. ullaRequestCmd()

SYNTAX

```
ULLA_API ullaResultCode ullaRequestCmd (IN CmdDescr_t *cmddescr, IN
    handleAsyncCmd_t handler, OUT CmdId_t *cmdId)
```

DESCRIPTION

Send asynchronous commands to Link or Link Provider through ULLA Core. This method can be called by an LU to request a command to be executed asynchronously on a specific Link or LP. Upon completion of each command execution, the callback specified is called.

ARGUMENTS

The cmddescr provides information about the command, the handler is the pointer to the callback function and cmdId is the returned identifier assigned by ULLA Core for the current command request.

RETURNS

ullaRequestCmd() will return ULLA_OK if the command request was accepted, it will return ULLA_ERROR_OUT_OF_MEMORY if the queue for asynchronous commands is full, it will return ULLA_ERROR_INVALID_COMMAND if the command cannot be executed by link or LP,it will return ULLA_ERROR_INVALID_CLASS if there is an invalid classname in cmddescr, it will return ULLA_ERROR_UNKNOWN_ID if there is an invalid link/lp ID in cmddescr, it will return ULLA_ERROR_CMD_NOT_ALLOWED if the LU does not have sufficient privileges to execute the command, it will return ULLA_ERROR_ALREADY_LOCKED if there is already a lock on the link or LP, it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet, it will return ULLA_ERROR_INVALID_HANDLER if the pointer to the handler function is invalid, and it will return ULLA_ERROR_INVALID_PARAMETER if on of the parameters is wrong. Or if one of the required attribute values required to perform the command is not set properly then the error ULLA_ERROR_ATTRIBUTE_VALUE_INVALID will be returned. If the command cannot be completed for another reason the ULLA_ERROR_COMMAND_FAILED code is returned.

3.4.4. ullaCancelCmd()

SYNTAX

```
ULLA_API ullaResultCode ullaCancelCmd (IN CmdId_t cmdId)
```

DESCRIPTION

The ullaCancelCmd() method allows a LU to cancel an asynchronous command call previously requested with ullaRequestCmd().

ARGUMENTS

The cmdId identifies the command to be canceled, as returned by ullaRequestCmd().

RETURNS

The function will return ULLA_OK if the cancellation is successfully performed. It will return ULLA_ERROR_INVALID_COMMAND if the command with the requested cmdId does not exist, has already been cancelled, or has already expired and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.4.5. handleAsynCmd_t()

SYNTAX

```
typedef void(*handleAsyncCmd_t) (IN CmdId_t cmdId, IN ULLA_INT_t
    cmdRetVal)
```

DESCRIPTION

This is the type of callback function that must be provided by the LU when calling ullaRequestCmd().

ARGUMENTS

The parameter cmdId is the identifier of the command as returned by ullaRequestCmd() and the cmdRetVal is the return value of the command as returned by the LP.

3.5. ULLA reflection interface

The ULLA reflection interface enables the LU to retrieve information about the content of classes, offered commands, and details about each attribute. The reflection interface is an advanced feature and might not be supported by the ULLA Core. When the ULLA Core doesn't support the reflection interface the functions will return ULLA_ERROR_UNSUPPORTED_FEATURE. This is the only mandatory return value.

3.5.1. ullaGetSupportedClasses()

SYNTAX

```
ULLA_API ullaResultCode ullaGetSupportedClasses (IN Id_t lpId, OUT
    ULLA_STRING_t classList, INOUT ULLA_INT_t *size)
```

DESCRIPTION

The ullaGetSupportedClasses() function returns the list of classes supported by a Link Provider.

ARGUMENTS

The parameter lpId is the LP identifier, classList is the list of class names supported by the LP in a comma separated name list, and size is the length in bytes of the classList buffer. The length of the classList string is returned (without '\0' terminator). If size is set to 0 when calling the function the length needed for the string without '\0' terminator is returned.

RETURNS

The list of class names is encoded in a comma separated list of class names. The function will return ULLA_ERROR_UNSUPPORTED_FEATURE (mandatory) when the function is not supported, ULLA_OK if successfully completed, it will return ULLA_ERROR_BUFFER_TOO_SMALL if the classList buffer passed is too small. In such a case, size will return the required size of the buffer. The function will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong, it will return

ULLA_ERROR_UNKNOWN_ID if there is an invalid LP ID and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.5.2. ullaGetClassAttributes()

SYNTAX

```
ULLA_API ullaResultCode ullaGetClassAttributes (IN Id_t lpId, IN
    ULLA_STRING_t className, OUT ULLA_STRING_t attributeList, INOUT
    ULLA_INT_t *size)
```

DESCRIPTION

The ullaGetClassAttributes() function returns the attributes supported by a defined class.

ARGUMENTS

The parameter lpId is the LP identifier, className is the name of the targeted class, attributeList is the attribute list in a comma separated list of attribute descriptions. Each attribute description uses the following format: attributename:attributeType:attributeModifier. The parameter size is the length in bytes of the attributeList buffer. The length of the attributeList string is returned (without '\0' terminator). If size is set to 0 when calling the function the length needed for the string without '\0' terminator is returned.

RETURNS

The list of attributes is returned in a string built as a comma separated list of attribute descriptions. Each attribute description uses the following format: attributename:attributeType:attributeModifier, where attributename is made of the class name a dot and the attribute name. The attribute modifier can have the following values: RO, RW, WO.

The function will return ULLA_ERROR_UNSUPPORTED_FEATURE (mandatory) when the function is not supported, ULLA_OK if successfully completed, it will return ULLA_ERROR_BUFFER_TOO_SMALL if the attributeList buffer is too small. In such a case, size will return the required size of the buffer. The function will return ULLA_ERROR_INVALID_CLASS if the class name specified is not a valid class name for the LP, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong, it will return ULLA_ERROR_UNKNOWN_ID if there is an invalid LP ID and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.5.3. ullaGetCommandAttributes()

SYNTAX

```
ULLA_API ullaResultCode ullaGetCommandAttributes (IN Id_t lpId, IN
    ULLA_STRING_t className, IN ULLA_STRING_t commandName, OUT
    ULLA_STRING_t attributeList, INOUT ULLA_INT_t *size)
```

DESCRIPTION

The ullaGetCommandAttributes() function returns the list of attributes that have to be set up before calling a command.

ARGUMENTS

The parameter lpId is the LP identifier, className is the name of the targeted class, commandName is the name of the targeted command, attributeList is an array of bytes, where the list of attribute names is returned in a comma separated list of names, and size is the length in bytes of the attributeList buffer. The length of the attributeList string is returned (without '\0' terminator). If size is set to 0 when calling the function the length needed for the string without '\0' terminator is returned.

RETURNS

The function will return ULLA_ERROR_UNSUPPORTED_FEATURE (mandatory) when the function is not supported, ULLA_OK if completed successfully, it will return ULLA_ERROR_BUFFER_TOO_SMALL if the buffer passed is too small. In such a case, size will return the required size of the buffer. The function will return ULLA_ERROR_INVALID_CLASS if the class name specified is not a valid class name for the LP, it will return ULLA_ERROR_UNKNOWN_ID if there is an invalid LP ID, it will return ULLA_ERROR_INVALID_COMMAND if the command is unknown, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.5.4. ullaGetClassCommands()

SYNTAX

```
ULLA_API ullaResultCode ullaGetClassCommands (IN Id_t lpId, IN const
    ULLA_STRING_t className, OUT ULLA_STRING_t commandList, INOUT
    ULLA_INT_t *size)
```

DESCRIPTION

The ullaGetClassCommands() function returns the list of command supported by a class.

ARGUMENTS

The parameter lpId is the LP identifier, className is the name of the targeted class, commandList is the list of command names in a comma separated format, and size is the length in bytes of the commandList buffer.

When calling the function, size should be initialized to the length of the commandList buffer passed as parameter. The length of the commandList string is returned (without '\0' terminator). If size is set to 0 when calling the function the length needed for the string without '\0' terminator is returned.

RETURNS

The function will return ULLA_ERROR_UNSUPPORTED_FEATURE (mandatory) when the function is not supported, ULLA_OK if completed successfully, it will return ULLA_ERROR_BUFFER_TOO_SMALL if the commands buffer is too small. In such a case, size will return the required size of the buffer. The function will return ULLA_ERROR_INVALID_CLASS if the class name specified is not a valid class name for the LP, it will return ULLA_ERROR_UNKNOWN_ID if there is an invalid LP ID, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.5.5. ullaGetAttributeInfo()

SYNTAX

```
ULLA_API ullaResultCode ullaGetAttributeInfo (IN Id_t lpId, IN
    ULLA_STRING_t className, IN ULLA_STRING_t attributeName, OUT
    ULLA_STRING_t attributeDescription, INOUT ULLA_INT_t *size)
```

DESCRIPTION

The `ullaGetAttributesInfo()` function returns the attribute description for a specific attribute.

ARGUMENTS

The parameter `lpId` is the LP identifier, `className` is the name of the targeted class, `attributeName` is the name of the attribute, `attributeDescription` is the attribute description in the following format: `attributeName:unit:default_value:range_low:range_high:type:modifier:description`, where `attributeName` is made of the class name dot the attribute name. For example, "`ullaLink.rxBitRate:bits/s:0:0:0xffffffff:ULLA_INT:RO:Dowstream bit rate`", and `size` is the length in bytes of the attributeDescription buffer.

When calling the function, `size` should be initialized to the length of the `attributeDescription` buffer passed as parameter. The length of the `attributeDescription` string is returned (without '\0' terminator). If `size` is set to 0 when calling the function the length needed for the string without '\0' terminator is returned.

RETURNS

The function will return ULLA_ERROR_UNSUPPORTED_FEATURE (mandatory) when the function is not supported, `ULLA_OK` if completed successfully, it will return `ULLA_ERROR_BUFFER_TOO_SMALL` if the `attributeDescription` buffer is too small. In such a case, `size` will return the required size of the buffer. The function will return `ULLA_ERROR_INVALID_CLASS` if the class name specified is not a valid class name for the LP, it will return `ULLA_ERROR_UNKNOWN_ID` if there is an invalid LP ID, it will return `ULLA_ERROR_INVALID_ATTRIBUTE` if the attribute does not exist, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.6. ULLA error handling interface

The ULLA error handling interface enables the LU to retrieve a description of an error.

3.6.1. ullaGetErrorString()

SYNTAX

```
ULLA_API ullaResultCode ullaGetErrorString (OUT ULLA_STRING_t str,
    INOUT ULLA_INT_t *size)
```

DESCRIPTION

The `ullaGetErrorString()` method returns a null-terminated text string describing the last error occurred while calling methods belonging to the ULLA API. The ULLA Link User Library maintains a thread-specific error variable referring to the error code returned by the last non-successful function call; a subsequent call to ullaGetErrorString() therefore returns a brief description of the error that has occurred.

The allocation of the memory for the string buffer must be done in advance by the caller.

ARGUMENTS

The parameter str is the pointer to the buffer allocated by the caller where the error message is to be stored, and size is the length of the buffer allocated. The length of the result string is returned (without '\0' terminator). When size is set to 0 when calling the function the length needed for the string without '\0' teminator is returned.

RETURNS

The function will return ULLA_OK if the error string is correctly returned , it will return ULLA_ERROR_NO_KNOWN_ERRORS if there is no error, it will return ULLA_ERROR_BUFFER_TOO_SMALL if the buffer length is insufficient, and it will return ULLA_ERROR_INVALID_PARAMETERS if one of the parameters is wrong.

3.7. ULLA layer three configuration interface

The ULLA layer three (L3) configuration interface allows LUs to ask ULLA Core to trigger the layer three configuration and to request the Link identifier to be used to reach a certain layer three address. These are also advanged features and might not be supported by the ULLA Core.

3.7.1. ullaGetLinkIdFromDest()

SYNTAX

```
ULLA_API ullaResultCode ullaGetLinkIdFromDest (IN layer3Address_t
    *dest, OUT Id_t *linkId, OUT Id_t *lpId)
```

DESCRIPTION

This function provides the Link identifier that should be used to reach a certain layer three address. The ULLA implementation will use OS-specific services (e.g. consulting routing tables or querying a Connection Manager) to obtain this information.

ARGUMENTS

The parameter dest is the L3 address to reach, the linkId is the link identifier to be used to reach the passed L3 address, and the lpId is the LP identifier associated with the above linkId.

RETURNS

The function will return ULLA_OK if a link has been found, it will return ULLA_ERROR_DESTINATION_NOT_REACHABLE if the requested address cannot be reached by any link, it will return ULLA_ERROR_INVALID_PARAMETER if the given address is not valid, it will return ULLA_ERROR_UNSUPPORTED_FEATURE if the ULLA implementation does not support this feature, and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.7.2. ullaConfigureL3()

SYNTAX

ULLA_API ullaResultCode ullaConfigureL3 (IN Id_t linkId, IN layer3Address_t *dest)

DESCRIPTION

The `ullaConfigureL3()` function instructs the ULLA core to carry on OS-specific network services in order to reach the given layer three destination address through a given link.

ARGUMENTS

The parameter `linkId` is the link identifier the layer three configuration is requested for and `dest` is the layer three address to reach.

RETURNS

The function will return `ULLA_OK` if layer three set up has been completed successfully, it will return `ULLA_ERROR_UNSUPPORTED_FEATURE` if the ULLA implementation does not support this feature, it will return `ULLA_ERROR_DESTINATION_NOT_REACHABLE` if the requested layer three destination address is not reachable, it will return `ULLA_ERROR_INVALID_PARAMETER` if the given address is not valid, it will return `ULLA_ERROR_UNKNOWN_ID` if there is an invalid LP ID, it will return `ULLA_ERROR_COMMAND_NOT_ALLOWED` if the LU is not allowed to execute the function, it will return `ULLA_ERROR_ALREADY_LOCKED` if there is already a lock on the link or LP, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.8. ULLA historical tables interface

The ULLA historical tables interface allows LUs to create and manage historical tables that can be used for more detailed analysis of the ongoing communication and the surroundings. These are also advanged features and might not be supported by the ULLA Core.

3.8.1. ullaCreateHistoricalTable()

SYNTAX

ULLA_API ullaResultCode ullaCreateHistoricalTable (IN ULLA_STRING_t tableName, IN Id_t sourceId, IN ULLA_STRING_t valueName, IN ULLA_INT_t period, IN ULLA_INT_t count)

DESCRIPTION

The `ullaCreateHistoricalTable()` function allows the LU to generate a new historical table. The latest value of the given attribute is stored periodically as specified using `period` and `count`.

ARGUMENTS

The parameter `tableName` is the name given to the table, `sourceId` is the Link/LP identifier from which data is to be stored, `valueName` is the attribute or aggregator in format "`className.attributeName`", which should be collected, `period` is the periodicity of information storing in `ms`, and `count` is the maximum number of samples to be stored.

RETURNS

The function will return ULLA_ERROR_UNSUPPORTED_FEATURE (mandatory) when the function is not supported, `ULLA_OK` if completed successfully, it will return `ULLA_ERROR_INVALID_CLASS` if the wrong class is specified as part of `valueName` or if the specified `sourceId` does not exist, it will return `ULLA_ERROR_INVALID_ATTRIBUTE` if a wrong attribute is specified as part of `valueName`, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.8.2. ullaDeleteHistoricalTable()

SYNTAX

```
ULLA_API ullaResultCode ullaDeleteHistoricalTable (IN ULLA_STRING_t
    tableName)
```

DESCRIPTION

The `ullaDeleteHistoricalTable()` function allows the LU to delete a historical table.

ARGUMENTS

The parameter `tableName` is the name of the table, which should be deleted.

RETURNS

The function will return ULLA_ERROR_UNSUPPORTED_FEATURE (mandatory) when the function is not supported, `ULLA_OK` if completed successfully, it will return `ULLA_ERROR_INVALID_CLASS` if the wrong class is specified as part of `valueName` or if the specified `sourceId` does not exist, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.8.3. ullaToggleStatusHistoricalTable()

SYNTAX

```
ULLA_API ullaResultCode ullaToggleStatusHistoricalTable (IN
    ULLA_STRING_t tableName, OUT ULLA_INT_t status)
```

DESCRIPTION

The `ullaToggleStatusHistoricalTable()` function allows the LU to toggle the status of a historical table. If `status` is one, the table will be updated and if the table is full, the data collection will wrap around and continue with slot 0. In such a case old values will be overwritten. If `status` is zero the data collection is paused and further analysis can be carried out based on the latest content of the historical table.

ARGUMENTS

The parameter `tableName` id the name of the table, which status should be toggled, `status` is the status of activity after toggling (1=updating, 0=not updating).

RETURNS

The function will return ULLA_ERROR_UNSUPPORTED_FEATURE (mandatory) when the function is not supported, `ULLA_OK` if completed successfully, it will return `ULLA_ERROR_INVALID_CLASS` if the wrong class is specified as part of `valueName` or if the specified `sourceId` does not exist, it will return `ULLA_ERROR_INVALID_PARAMETER` if one of the parameters is wrong, and it will return `ULLA_ERROR_NOTREGISTERED` if the LU is not registered, yet.

3.9. ULLA Link Manager interface

The ULLA Link Manager interface includes all functions that are related to the LM functionality.

`LmAuthorizationHandlers_t` struct

This is the data structure passed from the Link Manager to the ULLA Core upon registration with `registerLm()`. It consists of several function pointers that are described in more detail in the following.

3.9.1. lmRegisterLu()

SYNTAX

```
ullaResultCode(*LmAuthorizationHandlers_t::lmRegisterLu)  (IN  LuId_t
    luId,  IN  ULLA_INT_t  privilegeLevel,  IN  ullaApplicationID_t
    appId)
```

DESCRIPTION

This is the function called by the ULLA Core to register a LU with the LM.

ARGUMENTS

The parameter `luId` is the identifier of the LU, which the LM can store internally, the `privilegeLevel` gives the privilege level the LU requests, and `appId` is the unique application identifier, which the LM can use to identify the application. The `appId` is calculated in a platform dependent way.

RETURNS

The function will return `ULLA_AUTHORIZATION_OK` if the LU is allowed to register, `ULLA_AUTHORIZATION_FAILED` if the LU is not allowed to register.

3.9.2. lmDeregisterLu()

SYNTAX

```
ullaResultCode(*LmAuthorizationHandlers_t::lmDeregisterLu) (IN LuId_t
    luId)
```

DESCRIPTION

The `lmDeregisterLu()` function is used in order to deregister an LU from the LM.

ARGUMENTS

The parameter `luId` is the identifier of the link user to be deregistered.

RETURNS

The function will return `ULLA_AUTHORIZATION_OK` if deregistration is successful and it will return `ULLA_AUTHORIZATION_FAILED` if deregistration did not succeed.

3.9.3. lmCommandAuthorise()

SYNTAX

```
ullaResultCode(*LmAuthorizationHandlers_t::lmCommandAuthorize)    (IN
    LuId_t  luId,  IN  ULLA_INT_t  privilegeLevel,  IN  CmdDescr_t
    *cmdDescr, OUT ullaResultCode *result)
```

DESCRIPTION

The `lmCommandAuthorize()` function is used for authorizing commands.

ARGUMENTS

The parameter `luId` is the identifier of the LU, `privilegeLevel` is the current privilege level of the LU, which requested the command, the `cmdDescr` described the command the LU wants to execute, and the `result` is the `ullaResultCode` of the command when the LM has performed the operation.

RETURNS

The function will return `ULLA_AUTHORIZATION_OK` if the LU is allowed to perform a command, it will return `ULLA_AUTHORIZATION_FAILED` if the LU is not allowed to perform a command, and it will return `ULLA_OPERATION_PERFORMED` if the LM has executed the command, the result is stored in the `result` parameter.

3.9.4. lmSetAttributeAuthorize()

SYNTAX

```
ullaResultCode(*LmAuthorizationHandlers_t::lmSetAttributeAuthorize)
    (IN LuId_t luId, IN ULLA_INT_t privilegeLevel, IN AttrDescr_t*
    attrDescr, OUT ullaResultCode *result)
```

DESCRIPTION

The `lmSetAttributeAuthorize()` function is used to authorize a set attribute operation.

ARGUMENTS

The parameter `luId` is the identifier of the LU, `privilegeLevel` is the current privilege level of the LU, the `attrDescr` is the attribute that has to be changed, and the `result` is the `ullaResultCode` of the `ullaSetAttribute` function when the LM has performed the operation

RETURNS

The function will return `ULLA_AUTHORIZATION_OK` if the LU is allowed to change an attribute, it will return `ULLA_AUTHORIZATION_FAILED` If the LU is not allowed to change an attribute, and it will return `ULLA_OPERATION_PERFORMED` if the LM has set the attribute, the result is stored in the `result` parameter.

3.9.5. lmRequestInfoAuthorise()

SYNTAX

```
ullaResultCode(*LmAuthorizationHandlers_t::lmRequestInfoAuthorize)
    (IN LuId_t luId, IN ULLA_INT_t privilegeLevel, IN ULLA_STRING_t
    query, OUT ullaResult_t *result)
```

DESCRIPTION

The `lmRequestInfoAuthorize()` function is used to authorize a query.

ARGUMENTS

The parameter `luId` is the identifier of the LU, `privilegeLevel` is the current privilege level of the LU, `query` is the query that has to be executed, and `result` is the `ullaResult` of the query when the LM has performed the operation.

RETURNS

The function will return `ULLA_AUTHORIZATION_OK` if the LU is allowed to request a piece pf information on the Link, it will return `ULLA_AUTHORIZATION_FAILED` if the LU is not allowed to request a piece of information on the Link, and it will return `ULLA_OPERATION_PERFORMED` if the LM has performed the query, the result is stored in the `result` parameter.

3.9.6. lmRequestNotificationAuthorize()

SYNTAX

```
ullaResultCode(*LmAuthorizationHandlers_t::lmRequestNotificationAutho
    rize) (IN LuId_t luId, IN ULLA_INT_t privilegeLevel, IN
    ULLA_STRING_t query)
```

DESCRIPTION

The `lmRequestNotificationAuthorize()` function is used to authorize a notification request. The LM will never perform such a notification request itself.

ARGUMENTS

The parameter `luId` is the identifier of the LU, `privilegeLevel` is the current privilege level of the LU, and `query` is the query for the notification request.

RETURNS

The function will return `ULLA_AUTHORIZATION_OK` if the LU is allowed to request a notification for the Link, and it will return `ULLA_AUTHORIZATION_FAILED` if the LU is not allowed to request a notification for the Link.

3.9.7. lmPrepareCmd()

SYNTAX

```
ullaResultCode(*LmAuthorizationHandlers_t::lmPrepareCmd) (IN LuId_t
    luId, IN ULLA_INT_t privilegeLevel, IN CmdDescr_t *cmddescr)
```

DESCRIPTION

The `lmPrepareCmd()` function is used to authorize a Link lock.

ARGUMENTS

The parameter `luId` is the identifier of the LU, `privilegeLevel` is the current privilege level of the LU, the `cmdDescr` describes the command the LU wants to execute.

RETURNS

The function will return `ULLA_AUTHORIZATION_OK` if the LU is allowed to lock the Link and it will return `ULLA_AUTHORIZATION_FAILED` if the LU is not allowed to lock the Link.

3.9.8. lmConfigureL3()

SYNTAX

```
ullaResultCode(*LmAuthorizationHandlers_t::lmConfigureL3) (IN LuId_t
    luId, IN ULLA_INT_t privilegeLevel, IN Id_t linkId, IN
    layer3Address_t *dest)
```

DESCRIPTION

The lmConfigureL3() function is used to authorize a configuration request for the layer three setup.

ARGUMENTS

The parameter luId is the identifier of the LU, privilegeLevel is the current privilege level of the LU, linkId is the Link identifier the layer three configuration is requested for, and dest gives the layer three address to reach.

RETURNS

The function will return ULLA_AUTHORIZATION_OK if the LU is allowed to perform lmConfigureL3(), it will return ULLA_AUTHORIZATION_FAILED if the LU is not allowed to perform lmConfigureL3(), and it will return ULLA_OPERATION_PERFORMED if the LM has performed the configuration.

3.9.9. ullaGetAppInfo()

SYNTAX

```
ULLA_API ullaResultCode ullaGetAppInfo (IN LuId_t luId, OUT
    ULLA_STRING_t info, INOUT ULLA_INT_t *size)
```

DESCRIPTION

The Link Manager uses the ullaGetAppInfo() function in order to retrieve information about an application from the ULLA Core.

ARGUMENTS

The parameter luId is the LU identifier, returned upon registration time, the info a string describing the application, and size is the length of the buffer allocated. The length of the result string is returned (without '\0' terminator). When size is set to 0 when calling the function the length needed for the string without '\0' terminator is returned.

RETURNS

The function will return ULLA_OK if successfully completed, ULLA_AUTHORIZATION_FAILED when the LU is not allowed to access this information, it will return ULLA_ERROR_INVALID_LUID if the luId is not valid, it will return ULLA_ERROR_INVALID_PARAMETER if one of the parameters is wrong, and it will return ULLA_ERROR_NOTREGISTERED if the LU is not registered, yet.

3.10. ULLA Query Language

The ULLA Query Language is a well-defined subset of the SQL designed for providing an accessible data base abstraction at the Core of ULLA approach, while still being simple to implement. In essence, only queries of the form

SELECT <attributes> FROM <classes> WHERE <conditions>

are allowed, where attributes and classes are comma separated lists of attribute and class names, and the *optional* WHERE-clause contains usual comparison operators, numbers, and references to attributes. Joined queries are supported, so <classname>.<attributename> is a valid UQL attribute name. All SQL aggregators (MAX, MIN, AVG, SUM, COUNT) are likewise supported. For more detailed introduction to the use of UQL in queries and notifications, see deliverables D2.4 “Final Architecture and API” and D3.4 “API Guidebook”.

The complete Backus-Naur form of the ULLA Query Language is as follows:

```
<uql-statement> ::= <select-statement>

<select-statement> ::= “SELECT” <select-clause> “FROM” <from-clause> [ <where-clause> ]

<select-clause> ::= <attribute-list> | “*” | <aggregator> “(*)” | <aggregator> “(“ <attribute-list> “)”

<attribute-list> ::= <attribute-list> “,” <attribute-ref> | <attribute-ref>

<attribute-ref> ::= <attribute> | <table-ref> “.” <attribute>

<aggregator> ::= “MAX” | “MIN” | “AVG” | “SUM” | “COUNT”

<attribute> ::= <name> | <name> “_” <qualifier>

<qualifier> ::= “UNDEFINED” | “HARDCODED” | “THEORETICAL” | “ESTIMATED” |
                “MEASURED” | “EXACT”

<from-clause> ::= <tables>

<tables> ::= <tables> “,” <table-ref> | <table-ref>

<table-ref> ::= <name>

<where-clause> ::= “WHERE” <search-condition>

<search-condition> ::= <search-condition> “OR” <search-condition> |
                       <search-condition> “AND” <search-condition> |
                       “NOT” <search-condition> | “(“ <search-condition> “)” | <predicate>

<predicate> ::= <comparison-predicate>

<comparison-predicate> ::= <scalar-exp> <comparison> <scalar-exp>

<scalar-exp> ::= <scalar-exp> “+” <scalar-exp> | <scalar-exp> “-” <scalar-exp> |
                 <scalar-exp> “*” <scalar-exp> | <scalar-exp> “/” <scalar-exp> |
                 “+” <scalar-exp> | “-“ <scalar-exp> | <const-ref> | <attribute-ref> | “(“ <scalar-exp> “)”

<const-ref> ::= <integer> | <float>

<comparison> ::= "=" | "<" | ">" | "<>" | "!=" | "<=" | ">="

<name> ::= <letter> { <letter> }

<integer> ::= <digit> { <digit> }

<float> ::= <integer> | <integer> “.” <integer>
```

Finally, <digit> contains digits “0”..”9”, and <letter> consists of lower- and uppercase ASCII letters “a”-“z” and “A”-“Z”.

4. Link Provider interface

The Link Provider Interface forms the lower part of the Unified Link-Layer API. It is composed of three sets of entry points:

1. Methods used by the ULLA Core to initialise/deinitialise Link Provider components. These function calls will only be used in a specific type of ULLA implementation where ULLA Core initiates the loading of the Link Providers. Other types of ULLA Core implementations that make use of some external system mechanism (e.g. hotplugging system in Linux) to load the Link Providers would not require use of these methods.
2. Methods exported by the ULLA Core to the Link Provider.
3. Methods exported by the Link Provider to the ULLA Core.

4.1. Function calls for loading Link Provider

The first set is composed by the two following functions:

4.1.1. lpInit()

SYNTAX

```
lpResultCode lpInit(UepIf_t* UepIf);
```

DESCRIPTION

Each Link Provider must export a symbol of this type with name "lpInit". This function must be called by the ULLA Core upon loading of a Link Provider. Since all Link Providers are exporting this symbol, the ULLA Core implementation must explicitly retrieve the address of the lpInit() function for each Link Provider that has been loaded using OS specific means.

ARGUMENTS

The structure UepIf* passed as an argument carries the methods exported by the core to the Link Provider.

RETURNS

LP_OK on success.

4.1.2. lpTerm()

SYNTAX

```
lpResultCode lpTerm(void);
```

DESCRIPTION

Each Link Provider must export a symbol of this type with name "lpTerm". This function must be called by the ullaCore in order to allow the Link Provider to execute the proper termination routines (e.g. to call unregisterLp()) before the Link Provider library itself is unloaded. This call takes no arguments.

ARGUMENTS

None.

RETURNS

LP_OK on success.

4.2. ULLA Event Interface

The second set of entry points represent the interface exported by the ULLA Core towards the Link provider. It is composed by the following five function pointers:

4.2.1. handleEvent()

SYNTAX

```
void (*handleEvent)(IN RuId_t ruId, AttrDescr_t* attrdescr );
```

DESCRIPTION

The "handleEvent" method is the Ulla Core default callback function for handling events generated by Link Providers. This method is exported by the Ulla Core to handle event reports from link providers, which have been requested with requestUpdate().

ARGUMENTS

The RuId_t argument is the Id of the requestUpdate call, which is being answered. If the ruId is set to zero, the event is generated autonomously by the LP and represents an unsolicited update. The AttrDescr_t argument is the descriptor of the reported attribute, containing the new value. **RETURNS**

Void function.

4.2.2. registerLp()

SYNTAX

```
lpResultCode (*registerLp)(IN LpDescr_t* lpDescr, OUT Id_t* lpId);
```

DESCRIPTION

The "registerLp" method is exported by the Ulla Core and must be called by each Link Provider upon startup, in order to be registered within the ULLA system.

ARGUMENTS

The LpDescr_t attribute is a pointer to a structure providing Link Provider information and methods. Id_t argument is the returned numeric identifier of the Link Provider.

RETURNS

ULLA_OK on success.

ULLA_ERROR_VERSION_MISMATCH if the ULLA version is not compatible to the one requested by the Link Provider.

4.2.3. unregisterLp()

SYNTAX

```
lpResultCode (*unregisterLp)(IN Id_t lpId);
```

DESCRIPTION

The “unregisterLp” method allows a Link Provider to detach from the ULLA system.

ARGUMENTS

The Id_t argument contains the numeric identifier of the Link Provider to be detached.

RETURNS

ULLA_OK on success

ULLA_ERROR_UNKNOWN_ID if the Link Provider ID does not exist or has already been unregistered.

4.2.4. registerLink()

SYNTAX

```
lpResultCode (*registerLink) (IN Id_t lpId, OUT Id_t* linkId);
```

DESCRIPTION

The “registerLink” method allows a Link Provider to register a new Link to the Ulla Core. This method is to be called by the Link Provider if new Links are found after executing a scanAvailableLinks command. The method must be invoked once for each new Link discovered; the registration of a new link causes the creation of a new instance of the Link class. The type of the class is determined upon LP registration through registerLp(). The attribute values of the newly created Link are not passed at registration time; instead, they are inserted and updated only by explicit calls to getAttribute() or when reporting an event through handleEvent().

ARGUMENTS

The Id_t argument is the identifier of the Link Provider making the call. This identifier is the one returned by registerLp(). The Id_t argument is the identifier of the new Link as determined by the Ulla Core.

RETURNS

ULLA_OK on success.

ULLA_ERROR_FAILED register failed.

4.2.5. unregisterLink()

SYNTAX

```
lpResultCode (*unregisterLink)(IN Id_t linkId);
```

DESCRIPTION

The “unregisterLink” method must be called after executing a scanAvailableLinks command if some Link, which was previously registered with registerLink(), no more exists. As a consequence of this call, the corresponding instance of the Link class is deleted from ULLA Storage, along with all its attributes.

ARGUMENTS

The Id_t argument is the identifier of the link to be unregistered.

RETURNS

ULLA_OK on success.

ULLA_ERROR_FAILED unregister failed.

4.2.6. registerChannel()

SYNTAX

```
ullaResultCode (*registerChannel)(IN UcDescr_t *ucDescr, OUT Id_t
*ucId);
```

DESCRIPTION

This method is used to register a generic Channel object with the ULLA Core.

ARGUMENTS

The parameter ucDescr is the pointer to the Channel descriptor. The parameter ucId is the unique identifier that the Ulla Core assigns to the Channel.

RETURNS

ULLA_OK on success.

ULLA_ERROR_INVALID_DESCRIPTOR if an error is found in the descriptor.

ULLA_ERROR_UNSUPPORTED_FEATURE is the Core does not support this.

4.2.7. unregisterChannel()

SYNTAX

```
ullaResultCode (*unregisterChannel)(IN Id_t ucId);
```

DESCRIPTION

This method is used to unregister a generic Channel object with the ULLA Core when it is no longer available, for instance if the device is turned off.

ARGUMENTS

The parameter ucId is the unique identifier that the Ulla Core assigned to the Channel.

RETURNS

ULLA_OK on success.

ULLA_ERROR_UNKNOWN_ID if the identifier is not found.

ULLA_ERROR_UNSUPPORTED_FEATURE is the Core does not support this.

4.2.8. mapChannel()

SYNTAX

```
ullaResultCode (*mapChannel)(IN Id_t ucId, IN Id_t ulId, IN
ChannelRelationship_t relationship);
```

DESCRIPTION

This method is used to map an already registered Channel object to a Link object. This mapping identifies the relationship between the Link and Channel.

ARGUMENTS

The parameter ucId is the unique identifier that the Ulla Core assigned to the channel. The parameter ulId is the unique identifier that the Ulla Core assigned to the link. The parameter relationship indicates whether the link shares, exclusively uses or is not allowed to use the Channel.

RETURNS

ULLA_OK on success.

ULLA_ERROR_UNKNOWN_ID if the identifier is not found.

ULLA_ERROR_UNSUPPORTED_FEATURE is the Core does not support this.

4.2.9. unmapChannel()

SYNTAX

```
ullaResultCode (*unmapChannel)(IN Id_t ucId, IN Id_t ulId, IN
ChannelRelationship_t relationship);
```

DESCRIPTION

This method is used to unmap an already mapped Channel object with a Link object.

ARGUMENTS

The parameter ucId is the unique identifier that the Ulla Core assigned to the Channel. The parameter ulId is the unique identifier that the Ulla Core assigned to the Link. The parameter relationship indicates whether the link shares, exclusively uses or is not allowed to use the channel.

RETURNS

ULLA_OK on success.

ULLA_ERROR_UNKNOWN_ID if the identifier is not found.

ULLA_ERROR_NO_MAPPING if there is no mapping matching this one.

ULLA_ERROR_UNSUPPORTED_FEATURE is the Core does not support this.

4.3. Link Provider provided Interface

The third set of entry points represents the interface exported by a Link Provider towards the ULLA Core and is composed by the following seven function pointers:

4.3.1. getAttribute()

SYNTAX

```
lpResultCode (*getAttribute) (INOUT AttrDescr_t* attDescr);
```

DESCRIPTION

This method allows the ULLA Core to retrieve an attribute from a Link Provider or Link.

ARGUMENTS

The AttrDescr_t parameter is the descriptor for the requested attribute

RETURNS

LP_OK on success.

LP_ERROR_INVALID_ATTRIBUTE the attribute name is not valid.

LP_ERROR_BAD_PARAMETER if something is incorrect in the provided attribute descriptor e.g. wrong type, value out of range.

LP_ERROR_UNKNOWN_ID if the Link id requested is unknown to the LP.

4.3.2. setAttribute()

SYNTAX

```
lpResultCode (*setAttribute) (IN AttrDescr_t* attDescr);
```

DESCRIPTION

This method allows the ULLA Core to set a Link or Link Provider attribute.

ARGUMENTS

The AttrDescr_t parameter contains the description of such attribute.

RETURNS

LP_OK on success.

LP_ERROR_INVALID_ATTRIBUTE the attribute name is not valid.

LP_ERROR_BAD_PARAMETER if something is incorrect in the provided attribute descriptor e.g. wrong type, value out of range.

LP_ERROR_UNKNOWN_ID if the Link or Link Provider id requested is unknown to the LP.

4.3.3. freeAttribute()

SYNTAX

```
void (*freeAttribute) (IN AttrDescr_t* attDescr);
```

DESCRIPTION

This method is used to free a result allocated by a getAttribute() call.

ARGUMENTS

The AttrDescr_t parameter is a pointer to the result to be freed.

RETURNS

LP_OK on success

4.3.4. execCmd()

SYNTAX

```
lpResultCode (*execCmd) (IN CmdDescr_t* cmdDescr);
```

DESCRIPTION

The execCmd method is called by the Ulla Core to request the Link Provider to execute a command. A Link Provider is only capable of executing one command at the same time.

ARGUMENTS

The CmdDescr_t argument contains the description of the command to be executed.

RETURNS

LP_OK on success.

LP_ERROR_BAD_COMMAND if the requested command is not supported.

LP_ERROR_BAD_PARAMETER if the supplied parameters are not correct.

LP_ERROR_UNKNOWN_ID if the Link id requested does not exist.

LP_ERROR_COMMAND_AFFECTS_MULTIPLE_LINKS, the command has been requestd on a specific link but it impacts all links associated with the link provider. The command is not executed. ULLA can decide to give up or re-issue the command with linkID = 0.

LP_ERROR_COMMAND_FAILED, the command has for some reason not been performed.

LP_ERROR_ALREADY_EXECUTING_CMD is the LP or Link is already executing a command.

4.3.5. cancelCmd()

SYNTAX

```
lpResultCode (*cancelCmd) (Id_t id);
```

DESCRIPTION

The cancelCmd method is called by the Ulla Core to request the Link Provider or Link to stop execution of a command. As a LP can only execute one command at the same time cancelCmd will always cancel the command that is currently being executed.

ARGUMENTS

The Id_t argument contains the id of the Link/LP executing the command to be stopped.

RETURNS

LP_OK on success , LP_ERROR_NO_CMD_TO CANCEL if the command has already completed.

LP_ERROR_UNKNOWN_ID if the Link id requested does not exist.

LP_ERROR_COMMAND_FAILED, the command cannot be stopped.

4.3.6. requestUpdate()

SYNTAX

```
lpResultCode (*requestUpdate) (IN RuId_t ruId, IN RuDescr_t* ruDescr,
IN AttrDescr_t* attrDescr);
```

DESCRIPTION

This method is used by the ULLA Core to request a notification from the Link Provider in response to link events.

ARGUMENTS

The RuId_t argument is the numeric Id of the current requestUpdate call. The RuDescr_t argument is the request update descriptor associated to this requestUpdate() call. The AttrDescr_t argument is the descriptor of the attribute for which an update is requested.

RETURNS

LP_OK on success.

LP_ERROR_INVALID_ATTRIBUTE if the supplied attribute has not been recognised.

LP_ERROR_BAD_PARAMETER if the supplied parameters are not correct.

LP_ERROR_UNKNOWN_ID if the link id requested does not exist.

LP_ERROR_BAD_REQUEST_ID if the request update ID already exists.

4.3.7. cancelUpdate()

SYNTAX

```
lpResultCode (*cancelUpdate) (IN RuId_t ruId);
```

DESCRIPTION

By calling the cancalUpdate method, a previous call to requestUpdate() is canceled.

ARGUMENTS

The ruId argument contains the numeric Id of the current requestUpdate() call.

RETURNS

LP_OK on success.

LP_ERROR_BAD_REQUEST_ID if the request update ID does not exist.

4.3.8. getLpErrorString()

SYNTAX

```
lpResultCode (*getLpErrorString)(OUT ULLA_STRING_t str, IN ULLA_INT_t
len);
```

DESCRIPTION

The getLpErrorString() method returns a null-terminated text string describing the last error occurred while calling methods belonging to the Link Provider API.

ARGUMENTS

The ULLA_STRING_t argument is the pointer to the buffer allocated by the caller where the error message is to be stored. The ULLA_INT_t argument is the length of the buffer allocated.

RETURNS

LP_OK on success.

LP_ERROR_BAD_PARAMETER there is an error in the parameters.

LP_ERROR_NO_KNOWN_ERROR there is no known error.

LP_ERROR_NOT_ENOUGH_SPACE buffer is too small to store the error string.

4.3.9. getSupportedClasses()

SYNTAX

```
lpResultCode (*getSupportedClasses)(OUT ULLA_STRING_t classList,
INOUT ULLA_INT_t *size);
```

DESCRIPTION

The getSupportedClasses function returns the list of classes supported by a link provider.

ARGUMENTS

The list of class names is encoded in a comma separated list of class names. When calling the function, size should be initialized to the length of the classList array passed as parameter. When returning, size contains the size of the returned classList array.

RETURNS

LP_OK if successful.

LP_ERROR_NOT_ENOUGH_SPACE if the classList buffer passed is too small. In such a case, size will return the required size of the buffer.

4.3.10. getClassAttibutes()

SYNTAX

```
lpResultCode (*GetClassAttributes)(IN ULLA_STRING_t className,
ULLA_STRING_t attributeList, INOUT ULLA_INT_t *size);
```

DESCRIPTION

The getClassAttributes function returns the attributes supported by a defined class.

ARGUMENTS

The list of attributes is returned in a string built as a comma separated list of attribute description. The parameter className is the name of the targeted class. The parameter attributeList is the list of attribute list in a comma separated list of attribute description. The size parameter represents the length in bytes of the attributes list string.

RETURNS

LP_OK if successful.

LP_ERROR_NOT_ENOUGH_SPACE if the attributeList buffer is too small. In such a case, size will return the required size of the buffer.

LP_ERROR_INVALID_CLASS if the class name specified is not a valid class name for the link provider.

4.3.11. getCommandAttributes()

SYNTAX

```
lpResultCode  (*getCommandAttributes)(IN ULLA_STRING_t className, IN
ULLA_STRING_t   commandName,   OUT   ULLA_STRING_t   attributeList,
ULLA_INT_t *size);
```

DESCRIPTION

The getCommandAttributes function returns the list of attributes that have to be set up before calling a command.

ARGUMENTS

The parameter className is the name of the targeted class. The parameter commandName name is of the targeted command. The attributeList string contains the list of comma separated attribute names. The size parameter represents the length in bytes of the attributes list string.

RETURNS

LP_OK if successful.

LP_ERROR_NOT_ENOUGH_SPACE if the attributeList buffer is too small. In such a case, size will return the required size of the buffer.

LP_ERROR_INVALID_CLASS if the class name specified is not a valid class name for the link provider.

4.3.12. getClassCommands()

SYNTAX

```
lpResultCode  (*getClassCommands)(IN  ULLA_STRING_t  className,  INOUT
ULLA_STRING_t commandList, ULLA_INT_t *size);
```

DESCRIPTION

The getClasscommands function returns the list of command supported by a class.

ARGUMENTS

The parameter className is the name of the targeted class. The commandList parameter is the list of command names in a comma separated format and the parameter size is the length in bytes of the command list string.

RETURNS

LP_OK if successful.

LP_ERROR_INVALID_CLASS if the class is not supported.

LP_ERROR_NOT_ENOUGH_SPACE if the commands buffer is too small. In such a case, size will return the required size of the buffer.

LP_ERROR_INVALID_CLASS if the class name specified is not a valid class name for the Link Provider.

4.3.13. getAttributeInfo()

SYNTAX

```
lpResultCode (*getAttributeInfo)(IN ULLA_STRING_t className, IN
ULLA_STRING_t attributeName, OUT ULLA_STRING_t attributeDescription,
ULLA_INT_t *size);
```

DESCRIPTION

The getAttributeInfo function returns the attribute description for a specific attribute.

ARGUMENTS

The parameter className is the name of the targeted class. The parameter attributeName is the name of the attribute and the parameter attributeDescription is the description.

RETURNS

LP_OK if successful.

LP_ERROR_NOT_ENOUGH_SPACE if the attributeDescription buffer is too small. In such a

case, size will return the required size of the buffer.

LP_ERROR_INVALID_CLASS if the class name specified is not a valid class name for the link provider.

LP_ERROR_INVALID_ATTRIBUTE if the attribute does not exist.

LP_ERROR_INVALID_CLASS if the class is not supported.

4.3.14. getMeasureCap()

SYNTAX

```
lpResultCode (*getMeasureCap) (OUT UllaMeasureCap_t *measureCap);
```

DESCRIPTION

The getMeasureCap Function is used to get the measurement capabilities of the Link Provider.

ARGUMENTS

The parameter measureCap is the pointer to the capability linked list structure.

RETURNS

LP_OK if successful.

LP_ERROR_BAD_COMMAND the command is not known.

4.3.15. getStatisticsCap()

SYNTAX

```
lpResultCode (*getStatisticsCap) (OUT ULLA_STRING_t statisticsCap);
```

DESCRIPTION

The getStatisticsCap function is used to get the statistics capabilities of the Link Provider.

ARGUMENTS

The parameter statisticsCap contains a comma separated list of statistics that can be performed on measurements.

RETURNS

LP_OK if successful.

LP_ERROR_BAD_COMMAND the command is not known.

5. Low-end API

Among the various OSs for very small embedded devices such as wireless sensor nodes, TinyOS was chosen as an appropriate operating system, because it is an open-source environment designed for Wireless Wensor Networks (WSNs) with a very small footprint. It is a component-based operating system featuring an event-driven execution model. All of the TinyOS system libraries and applications are written in nesC programming language which is an extension of the standard C programming language. It is designed to support special needs of TinyOS: event-based concurrency model and the concept of components.
TinyOS executes only one program which consists of selected system components and custom components. Each component provides and uses interfaces which are collections of related functions. In practice, components usually declare functions in terms of interfaces. These interfaces are bi-directional: commands are used as downcalls to start the operation and events (or callbacks) are used as upcalls to signify when the operation is complete. The interfaces have to be defined in the configuration file. Furthermore, there is no separate compilation and runtime linking. Therefore, dynamic registration/deregistration of the LU and LP to the ULLA Core cannot be supported in TinyOS.

5.1. Link User Interface

In the following, the detailed interface definition has been listed. The functionality of most of the calls is the same as described in chapter 3 for the main API otherwise the differences are mentioned and explained.

The low end API is used for LUs running locally on embedded sensor nodes, also called Local LUs. Remote LUs, which instead run on gateway devices such as usual PCs with a gateway sensor node connected, will use the standard API.

```
typedef uint8_t ullaResult_t;


interface UCPIf {
   command ullaResultCode ullaSetAttribute(IN AttrDescr_t attrDescr);
   command ullaResultCode ullaPrepareCmd(IN CmdDescr_t* cmddescr);
   command ullaResultCode ullaDoCmd(IN CmdDescr_t* cmddescr,
         IN uint8_t timeoutValue);
   command ullaResultCode ullaRequestCmd(IN CmdDescr_t* cmddescr,
         IN handleAsyncCmd_t handler, OUT CmdId_t *cmdId);
   command ullaResultCode ullaCancelCmd(IN CmdId_t cmdId);
   event ullaResultCode handleAsyncCmd_t (CmdId_t cmdId,
         uint8_t cmdRetVal);
}
```

The interface describing the ULLA Command Processing is very similar to the standard ULLA. Some data types are different in order to minimize the implementation footprint. Additionally, one event was added. Following the TinyOS event-based architecture the event `handleAsyncCmd_t` will be signaled when the asynchronous command requested by `ullaRequestCmd` was completely performed. The name `handleAsyncCmd_t` was chosen to fit to the main ULLA although TinyOS does not support the principle of function pointers. The same reason applies for handleNotification_t.

```
interface UQPIf {
   command ullaResultCode ullaGetCoreDescriptor(INOUT CoreDescr_t
         *coreDescriptor);
   command ullaResultCode ullaRequestInfo(IN Query* query, OUT
         ullaResult_t *result);
   event ullaResultCode handleRequestInfo(IN ullaResult_t *result);
   command ullaResultCode ullaRequestNotification(OUT RnId_t* rnId,
         IN char *query, IN RnDescr_t* rndescr);
   command ullaResultCode ullaCancelNotification(IN RnId_t rnId);
   event ullaResultCode handleNotification_t (IN RnId_t rnId,
         IN ullaResult_t res, IN void* privdata);
   command ullaResultCode ullaResultNumFields(IN ullaResult_t res,
         OUT uint8_t *num);
   command ullaResultCode ullaResultNumTuples(IN ullaResult_t res,
         OUT uint8_t *num);
   command ullaResultCode ullaResultFieldNumber(IN ullaResult_t res,
         IN char *fieldName, OUT uint8_t *num);
   command ullaResultCode ullaResultValueLength(IN ullaResult_t res,
         IN uint8_t fieldNo, OUT uint8_t *length);
   command ullaResultCode ullaResultValueType(IN ullaResult_t res,
         IN uint8_t fieldNo, OUT BaseType_t *type);
   command ullaResultCode ullaResultValueToUint8(IN ullaResult_t res,
         IN uint8_t fieldNo, OUT uint8_t *value);
   command ullaResultCode ullaResultValueToUint16(IN ullaResult_t
         res, IN uint8_t fieldNo, OUT uint16_t *value);
}
```

The ULLA Query Processing is also only slightly adapted for the WSN-case. Again considering the event-based approach followed by TinyOS an event is used to realize the callback-function handleNotification_t. Also ullaRequestInfo is changed to a similar processing flow and an event was added that is signaled when the query was performed. One major reason is that the query processing might take a considerable amount of time especially when new measurements have to be performed: thus, an asynchronous implementation is advantageous. The accessor functions are adapted to the different set of data types used but the underlying semantics were not changed.

As the functionalities of Link Manager, layer three configuration, historical tables, and exception handling are not supported for the low-end API, the respective calls are not available.

5.2. Link Provider Interface

Similar to the Link User Interface, the low end API version of the Link Provider Interface has been described below. The main differences with respect to the main API are discussed.

```
interface linkProviderIf {
   command lpResultCode (*getAttribute) (IN AttrDescr_t* attDescr);
   command lpResultCode (*setAttribute) (IN AttrDescr_t* attDescr);
   command lpResultCode (*execCmd) (IN CmdDescr_t* cmdDescr);
   command lpResultCode (*requestUpdate) (IN RuId_t ruId,
         IN RuDescr_t* ruDescr, IN AttrDescr_t* attrDescr);
   command lpResultCode (*cancelUpdate) (IN RuId_t ruId);
}
```

The LP interface uses exactly the same calls and thus is only adapted to nesC syntax and defined as TinyOS interface.

```
interface UepIf {
   event void (*handleEvent)(IN RuId_t ruId, AttrDescr_t* attrdescr);
   event void (*handleGetAttribute) (IN AttrDescr_t* attDescr);
   event void (*handleSetAttribute) (IN AttrDescr_t* attDescr);
   event void (*handleExecCmd) (IN CmdDescr_t* cmdDescr);
   command ullaResultCode (*registerLink) (IN Id_t lpId, OUT Id_t*
         linkId);
   command ullaResultCode (*unregisterLink)(IN Id_t linkId);
}
```

The ULLA Event Processing is adapted to the event-based TinyOS architecture and extended by three events that enable the LP-component to signal that the performance of `getAttribute`, `setAttribute`, or `execCmd` was completed. The ULLA Core component can upon reception of such events access the new data and possibly forwards it to the LU. In contrast to the Link User interface the events signaling the completion of certain actions to the ULLA core are not part of the same interface. Instead, these are grouped in the `UepIf`. This design is based on the main ULLA because the function pointer to the `handleEvent` function is exchanged between LLA and ULLA Core during the dynamic registration process. Although such a dynamic registration process is not supported in TinyOS commands and events are still grouped in the same way.

As for the Link User Interface, some of the function calls available for the main ULLA are not available in the low-end API. The calls miss also mostly due to unsupported features, which are, e.g., the functions related to management of Channels and Measurement Capabilities.

6. Summary

This document describes the API Version 1.0 of the Unified Link Layer Application Programming Interface. The API consists of two parts: the LU Interface and LP Interface.

The LU Interface provides two basic types of function calls to send queries and commands to link providers and a number of other function calls to expose advanced services including Link Manager functionality, historical table management, class definition reflection, advanced error handling, L3 configuration and management. The LU Interface definition also accompanies a query language definition known as UQL. UQL, a sub-set of SQL is used in the LU function calls to specify queries and request notifications in a standard manner.

The LP Interface provides two basic types of function calls. One is exposed by the ULLA Core towards the Link Providers to handle events and LP registration and the other is exposed by the LP to the ULLA Core allowing the ULLA Core to query information and configure the Link providers and associated Links.

A separate Low-end API has also been defined here to realise ULLA services in resource limited embedded devices such as sensor devices. The low-end API provides the same set of basic function calls as the standard ULLA API, but without any function calls for advanced functionalities such as Link Manager, L3 configuration, reflection interface, etc.

Although API version 1.0 of the ULLA expects to provide a set of function calls addressing a number of common usage models and link technologies, a number of open issues needs to be investigated further. These issues will be resolved in the future versions of the API releases.

Appendix A Abbreviations

API	Application Programming Interface
CM	Connection Manager
DB	Database
L3	Layer three
LLA	Link Layer Adapter
LM	Link Manager
LP	Link Provider
LU	Link User
NIC	Network Interface Card
OS	Operating System
QoS	Quality of Service
SQL	Structured Query Language
UC	ULLA Core
UCP	ULLA Command Processing
UEP	ULLA Event Processing
ULLA	Unified Link Layer API
UQL	ULLA Query Language
UQP	ULLA Query Processing

Appendix B Data types

B.1 Types defined in ulla.h

B.1.1 Basic type definitions

In the following, the basic ULLA data type definitioins are listed. Since these are platform-dependent, only the Win32-version of the definitions is described here.

Name	Type	Description
ULLA_INT_t	INT32	Defines a standard 32-bit signed integer
ULLA_DOUBLE_t	double	Defines a standard 64-bit signed floating point number
ULLA_CHAR_t	CHAR	Defines an 8-bit character
ULLA_STRING_t	PCHAR	Defines a string type as a pointer to a char
ULLA_RAWDATA_t	PCHAR	Defines byte array as a pointer to a char

B.1.2 CmdDescr_t struct

This structure contains the name and the class of the command to be executed by the link/LP. It is used by ullaRequestCmd() or ullaDoCmd(). The table below shows the detailed content of the CmdDescr_t.

name	Type	Description
id	Id_t	Identifier of the Link or LP on which the command is to be executed.
className	ULLA_STRING_t	The class (e.g. ullaLink, ullaLinkProvider, 80211Link) the command belongs to.
cmd	ULLA_STRING_t	Null-terminated string containing the command to be executed.

B.1.3 AttrDescr_t struct

This data structure is used in two cases:

- for the request and the retrieval of link and LP attributes between the ULLA Event Processing and the LP, through getAttribute(), freeAttribute(), requestUpdate(), setAttribute() and handleEvent(). More details for these functions are given in chapter 4.
- for the setting of link or LP parameters through ullaSetAttribute().

Depending on which function the structure is passed to, the structure members act as IN or as OUT parameters.

Member name	Type	Description
id	Id_t	Identifier of the Link or LP for which the attribute is requested or reported.
className	ULLA_STRING_t	The name of the class the requested/reported attribute belongs to (e.g. ullaLink, ullaLinkProvider, 80211Link, etc.).
attribute	ULLA_STRING_t	The name of the attribute.
qualifier	AttrQual_t	The requested/reported qualifier of the attribute.
type	BaseType_t	The reported type of the attribute (ULLA_TYPE_INT, ULLA_TYPE_STRING...)
length	ULLA_INT_t	The length of the attribute in bytes. Used for types whose length is not known (e.g. ULLA_TYPE_STRING). Note: The length field is mainly useful for strings which cannot be null-terminated, for instance security keys, MAC addresses, link signatures, etc. Since the length field itself is unique, if multiple values are reported they must be of the same length. This is OK when the attribute has a well-defined length (e.g. a MAC address, or a MD5 hash), but problems may arise if a multiple-valued attribute needs a different length for each value.
numValues	ULLA_INT_t	How many values the attribute is composed of (this is to support multi-valued attributes).

data	void*	The pointer to the attribute value(s). This pointer is supposed to be an array of the type indicated in the apposite field (in other words, a pointer to data of the indicated type). The receiver of this data structure (the ULLA Core for getAttribute() and handleNotification(), the LP for setAttribute()) must explicitly cast the pointer to the appropriate type, e.g.
		switch(attr->type) {
		    case ULLA_TYPE_INT: {
		        int* value = (int*) attr->data;
		        for (i=0; i<attr->numValues; i++)
		            printf("Value = %d",value[i]);
		    } break;
		    case ULLA_TYPE_STRING: {
		        char** value = (char**) attr->data;
		        sprintf(format,
		        "Value = \%.%ds",attr->length);

		for (i=0; i<attr->numValues; i++)
		printf(format,value[i]);
		} break;
		}

B.1.4 CoreDescr_t struct

This data structure is used to retrieve version information of the ULLA Core through the `ullaGetCoreDescriptor()`. The version information is a string in the format `"MAJOR.MINOR"`.

Member name	Type	Description
`coreManufacturerName`	ULLA_CHAR_t [64]	Manufacturer of the ULLA Core
`coreVersion`	ULLA_CHAR_t [16]	ULLA Core version "`MAJOR.MINOR`"
`apiVersion`	ULLA_CHAR_t [16]	Version of the API implemented by the Core
`profile`	Profile_t	Bitmask of ULLA profiles supported

B.1.5 enum Profile_t

Describes the ULLA profiles supported by an ULLA Core.

Name	Constant Value	Description
ULLA_PROFILE_UNSPECIFIED	0	Supported profile is unspecified
ULLA_PROFILE_BASE	1	All the functionality described in ULLA base profile is supported
ULLA_PROFILE_EXTENDED	2	All the functionality described in the ULLA extended profile is supported
ULLA_PROFILE_HIGH_END	4	All the functionality described in the ULLA High profile is supported
ULLA_PROFILE_MAX	0x7fffffff	The maximum number of profiles supported.

B.1.6 enum LuRole_t

This enumeration lists different roles a LU can have when interacting with ULLA.

Member in enumeration	Description
ULLA_ROLE_LINK_MONITOR	The Link Monitor role: An application registering with this role will be able to use the UQP interface and optionally configure the measurement capability of LPs (if this is supported by the LP). Typically, such an application will monitor Link characteristics using the ULLA query and statistics calculation capabilities. The Link monitor cannot uypdate other attributes or issue commands that are not associated with Link or Chanel monitoring.
ULLA_ROLE_STD_LU	Standard Link User: An application registering with this role will only be able to use the UQP interface (i.e. ullaRequestInfo() and ulaRequestNotification() API calls) to access LP information (i.e. Cmd API service is not available within this role).
ULLA_ROLE_TRUSTED_LU	Trusted Link User: An application registering with this role will be able to use all standard UQP and UCP interfaces -- in other words API function calls: doCmd, requestCmd, setAttibute, requestInfo, requestNotification can all be used. Note that the way of determining the trust as well as implementing a suitable trust scheme is OS specific.
ULLA_ROLE_CM	Connection Manager role: An application registering with this role will be able to use the ULLA L3 (layer 3) configuration interface in addition to the standard UQP and UCP Interfaces. Typically, only a Connection Manager application would be required to register with this role.
ULLA_ROLE_PUSH_AGENT	Push agent (e.g. WAP) allows the creation of Links using the (createLink) method and also the setting of attributes associated with Links or Channels. This allows the population of the ULLA tables with information remotely obtained. One specific example is by an operator using a push agent to notify terminal devices when different link technologies are available.
ULLA_ROLE_LM	The Link Manager role: An application registering with this role will act as an authorization agent which intercepts and checks all the requests for commands and

	information passed to ULLA by different Link Users. It will then authorize or deny the requested service based on the ULLA service policy configuration. The ULLA LM interface will be used by this type of application.

B.1.7 enum BaseType_t

This enumeration lists basic data types used to classify data accessed via the accessor functions.

Member in enumeration	Description
ULLA_TYPE_INT	Interger
ULLA_TYPE_DOUBLE	Double
ULLA_TYPE_STRING	String
ULLA_TYPE_RAWDATA	Raws data bytes

B.1.8 enum AttrQual_t

This enumeration lists qualifiers that can be used to classify attributes.

Member in enumeration	Description
ULLA_QUAL_UNDEFINED	Undefined
ULLA_QUAL_HARDCODED	Hard-coded value
ULLA_QUAL_THEORETICAL	Theoretical value
ULLA_QUAL_ESTIMATED	Estimated value
ULLA_QUAL_MEASURED	Measured value
ULLA_QUAL_EXACT	Exact value

B.1.9 Id_t type definition

Type	Name	Description
ULLA_INT_t	Id_t	This identifier is determined by the ULLA Core and is unique for both Links and Link Providers

B.1.10 Summary of other enumerated type definitions

Complete type definitions could be found in the ulla.h header file.

Name	Type	Description

ullaResultCode	enum	Defines ULLA Core error codes -- see Appendix C
ullaExceptions	enum	Defined ULLA exceptions -- see B.3.2
lpResultCode	enum	Defines Link Provider error codes.
BaseType_t	enum	Define ULLA Base types -- see B.3.2
AttrQual_t	enum	Describes the Link Provider Attribute Qualifiers. -- see B.3.2
MediaType_t;	enum	Describes the wireless standard being used for the ullaLink.
MediaState_t;	enum	Defines the states an ullaLink can be in.
PowerModes_t;	enum	Defines the link provider power modes.
CostUnit_t	enum	Defines the Link cost unit.
MeasurementUnits_t	enum	Defines units in which measurement could be reported.
MeasurementType_t	enum	Defines categories for measurement types.
ChannelRelationship_t	enum	Defines possible relationships between Channels and Links.
MonitorType_t	enum	Defines methods how Channel monitoring can be performed.
ChannelType_t	enum	Defines all Channel types supported by the ullaChannel-functionality

B.2 Types defined in ullalp.h

B.2.1 RuId_t;

Identifies an update request made through the requestUpdate() Link Provider method.

Name	Type	Description
RuId_t	ULLA_INT_t	Request indentifier provided by the ULLA Core when an update request is made through requestUpdate()

B.2.2 struct LpDescr_t

This structure provides general information on the link provider, and contains the interface that the link provider must export to the ullaCore.

Name	Type	Description
apiVersion	ULLA_STRING_t	Version of the API implemented by the LP
lpIf	LpIf_t*	Pointer to a data structure containing methods supported

		by the LP

B.2.3 struct RuDescr_t

This structure conveys information for issuing an update request through requestUpdate().

Name	Type	Description
count	ULLA_INT_t	The maximum number of times the request update notification fires. If set to zero the request will remain until it is explicitely canceled by a cancelUpdate() call
period	ULLA_INT_t	The reporting interval for periodic notifications. For asynchronous notifications (i.e. event based) period will be set to zero.

B.2.4 struct UllaMeasureCap_t

An LP can optionally support measurements (and computation of their statistical values) with different measurement intervals and observation periods (windows). It is also possible that the LLA will support some configuration of update intervals and window. Therefore, in order for an application to get the full picture each LP (during registration) provides the capabilities to perform measurements and statistics. Then the LU can query on the measurement capabilities table (`ullaMeasureCap`) using UQL queries and set the window and interval (if permitted) using `ullaSetAttribute()`. The attributes are referenced by using a pseudo attribute name of the LP of : "`classname.attributename.configname`" where `classname` is the name of the class that the attribute corresponds to (such as `ullalink`), the `attributename` is the actual name of the measurement attribute (such as `noiselevel`) and `configname` is either interval or window depending on what capability being configured.

Name	Type	Description
Id	ULLA_INT_t	Identifier of the measurement capability
param	ULLA_STRING_t	Name of the attribute that is associated with this measurement capability
units	MeasurementUnits_t	Units of the measurement that is associated with this measurement capability
className	ULLA_STRING_t	The name of the class that this measurement attribute corresponds to
time	ULLA_INT_t	Time taken to perform a single measurement of this attribute in nanoseconds
power	ULLA_INT_t	Energy consumed to perform a single measurement of this attribute in nJ
accuracy	ULLA_INT_t	he absolute accuracy of a single measurement specified in the units associated with this measurement

precision	ULLA_INT_t	The precision (relative error) between measurements specified in the units associated with this attribute measurement
windowMax	ULLA_INT_t	The maximum window over which averaging (and other optional statistical operations) are performed for this attribute measurement. The window is specified in number of measurement samples of the attribute taken at intervals (detemined by the interval value). A value of 0 in the windowMin indicates that the window is a smoothing function using the formula x(n+1) = a*x(n) + (1-a)*s [where x(n) represents the previous stored smoothed values, s is the measurement and and a is the smoothing factor]. In which case the window contains the % value of a (i.e. 95%) +ve values indicate a periodic window with statistics computed on each non-overlapping window -ve values indicate a sliding window with statistics computed on each overlapping window that slides once per sample interval
windowMin	ULLA_INT_t	The minimum window over which averaging (and other optional statistical operations) are performed for this attribute measurement. The window is specified in number of measurement samples of the attribute taken at intervals (detemined by the interval value). A value of 0 in the windowMin indicates that the window is a smoothing function using the formula x(n+1) = a*x(n) + (1-a)*s [where x(n) represents the previous stored smoothed value, s is the measurement and and a is the smoothing factor]. In which case the window contains the % value of a (i.e. 95%) +ve values indicate a periodic window with statistics computed on each non-overlapping window -ve values indicate a sliding window with statistics computed on each overlapping window that slides once per sample interval
intervalMin	ULLA_INT_t	The minimum interval at which single measurements can be taken for this attrinbute meaurement. The interval (i) is specified in terms of either : Packet based measurements taken on every ith packet or beacon - denoted by +ve

		values. Time based to indicate time between successive meausrements (in milliseconds) - denoted by -ve values.
`intervalMax`	`ULLA_INT_t`	The maximum interval at which single measurements can be taken for thisattrinbute meaurement. The interval (i) is specified in terms of either : Packet based measurements taken on every ith packet or beacon - denoted by +ve values. Time based to indicate time between successive meausrements (in milliseconds) - denoted by -ve values
`interval`	`ULLA_INT_t`	The actual interval between successive measurements. Either :+ve values for packet intervals -ve values for time intervals (in milliseconds)
`window`	`ULLA_INT_t`	The actual window size in number of measurement samples. A value of 0 in the windowMin and windowMax indicates that the window is a smoothing function using the formula x(n+1) = a*x(n) + (1-a)*s [where x(n) represents the previous stored smoothed value, s is the measurement and and a is the smoothing factor]. In which case the window contains the % value of a (i.e. 95%) +ve values indicate a periodic window with statistics computed on each non-overlapping window -ve values indicate a sliding window with statistics computed on each overlapping window that slides once per sample interval
`nextCap`	`void *`	A pointer to the next capability structure in a NULL terminated linked list.

B.2.5 struct UcDescr_t

The ULLA Channel object contains a number of generic attributes that characterize the radio Channel and allow the Channel to be associated with ULLA Links. The Channel objects are created and updated by LPs in the same manner as Links, but using the functions `ullaRegisterChannel()` and `ullaUnregisterChannel()`.

Name	Type	Description
`technology[8]`	`MediaType_t`	The type of link technology the Channel is used for - could be multiple technologies (limited to 8)
`type[8]`	`ChannelType_t`	The type of channel this object corresponds to - could be multiple types for hybrid Channel

		concepts (limited to 8).
Bandwidth	ULLA_INT_t	The bandwidth of the channel in kHz
Frequency	ULLA_INT_t	The frequency of the channel in MHz
channelNumber[64]	Id_t[64]	The unique technology specific channel number for this channel - could be multiple numbers for aggregate channels (limited to 64). -ve numbers indicate not applicable
activityLevel	ULLA_INT_t	The activity level observed on the channel over the monitoring time as average number of packets / frames in monitoring window.
noiseLevel	ULLA_INT_t	The noiselevel objserved in dBm for transmissions detected on the channel as average noise level detected in monitoring window. Noise being the background (i.e. thermal, and receiver noise) detected when no packet transmissions are being detected.
monitorDuration	ULLA_INT_t	Duration of a single monitoring operation (in microseconds)
signalLevel	ULLA_INT_t	The average signal level in dBm of the detected packets that are observed over the monitoring window.
monitorMethod	MonitorType_t	The type of monitoring that is being performed on this channel. This can be passive (observation of all transmissions on the channel), active (observation of specific packets transmitted for monitoring purposes) or selective (only observing certain packet types).
extraData	ULLA_INT_t	Indicates whether optional configuration data is included
antennaConfig[256]	ULLA_CHAR_t	The antenna configuration specified as a byte array (limited to 256 bytes)
codes[256]	ULLA_CHAR_t	The codes used to specify the Channel configuration (for code division multiplexing or channel coding schemes) - specified as a bytes array (limited to 256 bytes).

B.2.6 struct LpIf_t

This structure provides the pointers to the method exported by the Link Provider to the ULLA Core.

```
    LpResultCode (*getAttribute) (INOUT AttrDescr_t* attDescr);
    LpResultCode (*setAttribute) (IN AttrDescr_t* attDescr);
    LpResultCode (*freeAttribute) (IN AttrDescr_t* attDescr);

    LpResultCode (*execCmd) (IN CmdDescr_t* cmdDescr);

    LpResultCode  (*requestUpdate)  (IN  RuId_t  ruId,  IN  RuDescr_t*
ruDescr, IN AttrDescr_t* attrDescr);

    LpResultCode (*cancelUpdate) (IN RuId_t ruId);

    LpResultCode   (*getLpErrorString)(OUT   ULLA_STRING_t   str,   IN
ULLA_INT_t len);
```

B.2.7 struct UepIf_

This structure contains the pointer to the Ulla Event Processing functions exported by the ULLA Core to each Link Provider. This structure is passed as a parameter when calling the LpInit() function to initialize a Link Provider. Details of the each function calls listed below could be found in section 4.3.

Name	Type	Description
Manufacturer	ULLA_STRING_t	Manufacturer of the ULLAcore
Version	ULLA_STRING_t	The version of the ULLA the ullaCore is compatible with
(*handleEvent)(IN RuId_t ruId, AttrDescr_t* attrdescr)	Void *	This method is exported by the ullaCore to handle envent reports from link providers, which have been requested with requestUpdate()
(*registerLp)(IN LpDescr_t* lpDescr, OUT Id_t* lpId)	ullaResultCode	This method is exported by the ullaCore and must be called by each Link Provider on startup, in order to be registered within ULLA.
(*unregisterLp)(IN Id_t lpId);	ullaResultCode	This method allows a Link Provider to be unregistered within ulla
(*registerLink) (IN Id_t lpId, OUT Id_t* linkId)	ullaResultCode	This method allows a Link Provider to register a new link to the ullaCore. This method is to be called by the link provider if new links are found after executing a scanAvailableLinks command. The method must be invoked once for each new link discovered, The registration of a new link causes the creation of a new instance of the link class. The type of the class is determined upon LP registration through registerLp(). The attribute values of the newly created link are not passed at registration time; instead, they are inserted and updated only by explicit calls to getAttribute() or when reporting an event through handleEvent().

(*unregisterLink)(IN Id_t linkId)	ullaResultCode	This method must be called after executing a scanAvailableLinks command if some link, which was previously registered with registerLink(), no longer exists. As a consequence of this call, the corresponding instance of the link class is deleted from ulla, along with all its attributes.
(*registerChannel)(IN UcDescr_t *ucDescr, OUT Id_t *ucId);	ullaResultCode	This method is used to register a generic Channel object with the ULLA Core.
(*unregisterChannel)(IN Id_t ucId)	ullaResultCode	This method is used to unregister a generic Channel object with the ULLA Core when it is no longer available. For instance if the device is turned off.
(*mapChannel)(IN Id_t ucId, IN Id_t ulId, IN ChannelRelationship_t relationship);	ullaResultCode	This method is used to map an already registered Channel object to a Link object. This mapping identifies the relationship between the Link and Channel.
(*unmapChannel)(IN Id_t ucId, IN Id_t ulId, IN ChannelRelationship_t relationship)	ullaResultCode	This method is used to unmap an already mapped Channel object with a link object

B.3 Types defined in ullalu.h

B.3.1 Identifiers : RnId_t, CmdId_t, LuId_t and ullaApplicationId_t

Name	Type	Description
RnId_t	ULLA_INT_t	Notification Request identifier returned from the ULLA core to the Link User upon call to requestNotification()
CmdId_t	ULLA_INT_t	Command request identifier returned from the ULLA core to the Link User upon call to requestCmd()
LuId_t	ULLA_INT_t	Link User identifier
ullaApplicationID_t[16]	ULLA_CHAR_t	application identifier generated by the ulla core using OS specific mechanisms, used with LM. The application identifier is a 16 bytes code

B.3.2 enum ullaExceptions

This enumerations lists exceptions that might occur during ULLA functions.

Member name	Description
`ULLA_EXCEPTION_NOTIFICATION_LIMIT_EXCEEDED`	Used to indicate that a Core limit on the number of outstanding notifications has been exceeded
`ULLA_EXCEPTION_MEMORY_LIMIT_EXCEEDED`	Used to indicate that a Core limit on the memory usage has been exceeded
`ULLA_EXCEPTION_PROCESSOR_LIMIT_EXCEEDED`	Used to indicate that a Core limit on the processor usage has been exceeded
`ULLA_EXCEPTION_DATABASE_CONNECTION_FAILED`	Used to indicate that the database connection has failed
`ULLA_EXCEPTION_CORE_SHUTDOWN`	Used to indicate that a Core is shutting down
`ULLA_EXCEPTION_CORE_FROZEN`	Used to indicate that a Core is suspending any new requests due to overload

B.3.3 Struct UllaExceptionDesc_t

This is the data structure used to describe an exception raised.

Member name	Type	Description
`id`	Id_t	The identifier of the link or LP object that resulted in the exception being raised.
`exception`	ullaExceptions	Enumeration of exceptions
`message`	ULLA_STRING_t	Some message describing the exception.

B.3.4 Struct LuDescr_t

This is the data structure passed from the LU to the ULLA core upon registration with `registerLu()`.

Member name	Type	Description
`Name`	`ULLA_STRING_t`	Name of the LU.
`Description`	`ULLA_STRING_t`	LU description (e.g. supplier, type of application, whatever).
`apiVersion`	`ULLA_STRING_t`	Lowest version of the ULLA API

		the LU is willing to use.
Profile	Profile_t	The requested profile type.
ullaExceptionHandler (IN UllaExceptionDesc_t *exceptionDesc)	void*	Handler for exceptions generated by the ULLA core.

B.3.5 enum layer3Protocol_t

This enumeration lists layer three address types that could be used during L3 configuration.

Member in enumeration	Description
ULLA_L3ADDR_IPV4	IP version 4 address
ULLA_L3ADDR_IPV6	IP version 6 address
ULLA_L3ADDR_IPX	IPX address
ULLA_L3ADDR_X25	X25 address
ULLA_L3ADDR_APPLETALK	Appletalk address

B.3.6 layer3Address_t struct

This data structure is used to describe a layer three address that might be configured or should be reached using a specific link.

The parameter protocol describes the L3 protocol that should be used for communication, the length gives the size of the address, and address gives the address itself.

Member Name	Type	Description
protocol	Layer3Protocol_t	The L3 protocol family
length	ULLA_INT_t	The length of significant data in the address buffer
Address[32]	ULLA_CHAR_t	The actual address buffer

B.3.7 struct LmAuthorizationHandlers_t

This is the data structure passed from the linkManager to the ullaCore upon registration with registerLm().

```
ullaResultCode (* lmRegisterLu)(IN LuId_t luId, IN ULLA_INT_t privilegeLevel, IN ullaApplicationID_t appId);

ullaResultCode (* lmDeregisterLu)(IN LuId_t luId);
ullaResultCode (* lmCommandAuthorize)(IN LuId_t luId, IN ULLA_INT_t
privilegeLevel, IN CmdDescr_t *cmdDescr, OUT ullaResultCode *result);

ullaResultCode (* lmSetAttributeAuthorize)(IN LuId_t luId, IN
ULLA_INT_t privilegeLevel, IN AttrDescr_t *attrDescr, OUT
ullaResultCode *result);

ullaResultCode (* lmRequestInfoAuthorize)(IN LuId_t luId, IN
ULLA_INT_t privilegeLevel, IN ULLA_STRING_t query, OUT ullaResult_t
*result);

ullaResultCode (* lmRequestNotificationAuthorize)(IN LuId_t luId, IN
ULLA_INT_t privilegeLevel, IN ULLA_STRING_t query);

ullaResultCode (* lmPrepareCmd)(IN LuId_t luId, IN ULLA_INT_t
privilegeLevel, IN CmdDescr_t* cmddescr);

ullaResultCode (* lmConfigureL3)(IN LuId_t luId, IN ULLA_INT_t
privilegeLevel, IN Id_t linkId, IN layer3Address_t *dest);
```

B.3.8 ullaResult_t;

Represents the identifier of a query result set, as returned by ullaRequestInfo()

Name	Type	Description
ullaResult_t	ULLA_INT_t	The purpose of this typedef is to hide the data structure to the Link User and to allow different kind of data representation in different implementations of the ULLA library. A numeric id is used (e.g. instead of a pointer) since the Link User should never access the result set by direct reference; instead, access is to be done using the apposite methods exported by the ULLA API. Memory allocation for the result set is done automatically by the ULLA library; on the other hand, memory deallocation is left to the Link User. As a consequence, after a call to requestInfo() the application must take care of freeing the memory allocated for the result set by calling the method ullaFreeResult() with the ullaResult identificator of the result set to be deallocated.

B.3.9 Struct RnDescr_t

This is the data structure used to describe a notification during registration.

Member name	Type	Description
count	ULLA_INT_t	Maximum number of notifications to be reported by calling the handler. If zero, the notification remains in force until explicitly canceled by ullaCancelNotification(). Otherwise the notification is automatically canceled after count reports.
Period	ULLA_INT_t	If this parameter is nonzero, the notification is periodic, and

		the value of the parameter represents the time interval in milliseconds between two periodic notifications. If the parameter is zero, the notification is event-driven.
privdata	void*	Private data that the application wants to be returned within the callback function.
query	ULLA_STRING_t	The query string (i.e. SELECT...).

Appendix C Error Codes

This enumeration defines error codes which are reported back as result by several ULLA functions. The same set of error codes is used for all whole LU interface.

Member in enumeration	Description
`ULLA_OK`	Operation successful.
`ULLA_ERROR_FAILED`	Operation failed, general failure, no specific reason.
`ULLA_ERROR_LIB`	Generic error in ULLA library, e.g. due to a bug.
`ULLA_ERROR_CORE`	Generic error in ULLA core, e.g. due to a bug.
`ULLA_ERROR_STORAGE`	Generic error in ULLA Storage system, e.g. in the internal database or in the external database implementation.
`ULLA_ERROR_API_VERSION_MISMATCH`	ULLA API version mismatch.
`ULLA_ERROR_SYNTAX_ERROR`	Syntax error e.g. in the query string or in the command string.
`ULLA_ERROR_INVALID_CLASS`	Invalid class in the query string or in the command string.
`ULLA_ERROR_INVALID_ATTRIBUTE`	Invalid attribute in the query string or in the command string
`ULLA_ERROR_UNSUPPORTED_FEATURE`	Requested feature is not supported, e.g. command not supported.
`ULLA_ERROR_INVALID_PARAMETER`	One or more of the function parameters are invalid.
`ULLA_ERROR_INVALID_ULLARESULT`	The `ullaResult_t` identifier does not exist or has already been deallocated.
`ULLA_ERROR_INVALID_FIELD`	Non-existing field number or field name.
`ULLA_ERROR_NOTREGISTERED`	The LU has not registered yet.
`ULLA_ERROR_LP_ERROR`	An error has occurred in the LP while executing the function call.
`ULLA_ERROR_BUFFER_TOO_SMALL`	The provided buffer size is insufficient, e.g. the buffer space allocated by the LU was not sufficient.
`ULLA_ERROR_NO_MORE_TUPLES`	All tuples in a result set have already been processed.
`ULLA_ERROR_NO_CURRENT_TUPLE`	This is returned when the LU is trying to access data within a result set without

	having called `ullaNextTuple()` at least once, or if the last call to `ullaNextTuple()` returned `ULLA_NO_MORE_TUPLES` but the LU is trying to access data anyway.
`ULLA_ERROR_TYPE_MISMATCH`	The requested field cannot be converted to the requested value, e.g. `ullaResultIntValue()` is called on a string field.
`ULLA_ERROR_NO_MORE_VALUES`	No more values are available for the current field. For an attribute field which contains N values, this error code is returned when a data access function (`ullaResultIntValue()`, `ullaResultStringValue()`, etc.) is called more than N times.
`ULLA_ERROR_UNKNOWN_ID`	The LP identifier does not exist or has already been unregistered.
`ULLA_ERROR_INVALID_LUID`	The LU identifier does not exist or has already been unregistered.
`ULLA_ERROR_UNSUPPORTED_PROFILE`	Returned upon a call to `ullaRegisterLu()`, if the LU requested support for a profile the ULLA core does not support.
`ULLA_ERROR_UNSUPPORTED_ROLE`	Returned upon a call to `ullaRegisterLu()`, if the LU requested a role the ULLA core does not support.
`ULLA_ERROR_ROLE_DENIED`	Returned upon a call to `ullaRegisterLu()`, if the ULLA core denies the requested role.
`ULLA_ERROR_LM_NOT_SUPPORTED`	Returned upon `ullaRegisterLm()` from an ULLA core that does not support external LM.
`ULLA_AUTHORIZATION_OK`	LM specific error codes: LM authorizes the operation.
`ULLA_AUTHORIZATION_FAILED`	LM specific error codes: LM does not authorize the operation.
`ULLA_OPERATION_PERFORMED`	LM specific error codes: LM has authorized and already performed the requested operation.
`ULLA_ERROR_INVALID_COMMAND`	Command cannot be executed by Link or LP.
`ULLA_ERROR_CMD_NOT_ALLOWED`	LU is not allowed to execute the command.

ULLA_ERROR_SETATTR_NOT_ALLOWED	LU is not allowed to set the attribute.
ULLA_ERROR_QUERY_NOT_ALLOWED	LU is not allowed to perform the requested query.
ULLA_ERROR_ALREADY_LOCKED	There is already a lock on the Link or LP.
ULLA_ERROR_PERIOD_TOO_SHORT	ULLA core or LP cannot handle the requested period.
ULLA_ERROR_ILLEGAL_HANDLER	Invalid pointer to handler used.
ULLA_ERROR_INVALID_NOTIFICATION	Invalid notification identifier.
ULLA_ERROR_NO_KNOWN_ERROR	There is no known error to return the error string.
ULLA_ERROR_ALREADY_REGISTERED	Indicates that the LU is already registered when a subsequent request is made.
ULLA_ERROR_NO_MAPPING	Used in `unmapChannel()` to indicate that there is no mapping set up for this relationship.
ULLA_ERROR_TIMEOUT	Used to indicate that a command (or other operation) has been timed out.
ULLA_ERROR_DESTINATION_NOT_REACHABLE	The provided address cannot be reached.
ULLA_ERROR_INVALID_QUALIFIER	Invalid qualifier being used
ULLA_ERROR_INVALID_VALUE	Trying to set an invalid value
ULLA_ERROR_SETATTR_NOTMULTIPLE	Trying to set multiple values for a single value attribute
ULLA_ERROR_SETATTR_READONLY	Trying to set a read only attribute
ULLA_ERROR_COMMAND_FAILED	Command failed to execute for a reason other than wrong attribute values
ULLA_ERROR_ATTRIBUTE_VALUE_INVALID	One of the attribute needed to execute a command is wrong

Appendix D Mandatory base classes

The mandatory ULLA classes, namely ullaLink and ullaLinkProvider classes are specified below.

D.1 UllaLink class

Attribute Name	Type	Description
bytesReceived	ULLA_INT_t	Number of bytes received. The attribute is only valid in state CONNECTED.
bytesSend	ULLA_INT_t	Number of bytes sent. The attribute is only valid in state CONNECTED.
coexistsWith	ULLA_INT_t	This is a list of ullalinks (i.e. IDs) that are allowed to coexist with this link, i.e. they can operate in parallel with no problems.
communicationMode	CommunicationMode_t	Communication mode used, examples: TxSIMPLEX, RxSIMPLEX, HALF-DUPLEX, FULL-DUPLEX
connectedTime	ULLA_INT_t	Connection time in milliseconds describing how long the connection exists without interruption. The connection time will be reset if a connection break occurs.
costPerUnit	ULLA_INT_t	Cost per defined unit
costUnit	CostUnit_t	Cost unit used. Examples: kByte, MByte, sec, hour, flat, etc.
degradingLink	ULLA_INT_t	This flag is set by the LLA using some technology-specific mechanism to indicate when the link is constantly degrading, e.g. in the case of moving further and further away from a cellular base station. To be interpreted as a boolean.
dependentOn	ULLA_INT_t	This is a list of ullalinks (i.e. IDs) that this link has some sort of dependency on. This attribute simply provides an indication that performance problems can arise when using this link with others the list. It does not attempt to quantify the possible interference levels.
excludes	ULLA_INT_t	This is a list of ullalinks (i.e. IDs) that are absolutely excluded from use when this link is in use.
frameReceiveErrors	ULLA_INT_t	Number of received frames that contained errors. The attribute is only valid in state CONNECTED.
frameSendErrors	ULLA_INT_t	Number of sent frames that could not be

Attribute Name	Type	Description
		transmitted successfully because of errors. The attribute is only valid in state CONNECTED.
id	Id_t	Unique link identifier
localL2-address	ULLA_STRING_t	Local link layer address
lpId	Id_t	Unique identifier of the Link Provider offering this link.
networkName Public	ULLA_STRING_t	Examples: ESSID for WLANs based on IEEE 802.11, mobile network code/country node
operatorName	ULLA_STRING_t	Examples: T-Mobile, Telefonica, etc.
packetsReceived	ULLA_INT_t	Number of packets received. The attribute is only valid in state CONNECTED.
packetsSent	ULLA_INT_t	Number of packets sent. The attribute is only valid in state CONNECTED.
plId	Id_t	Parent Link ID. If the link is an aggregate link the plId will be zero, otherwise the identifier of the respective aggregate link is given.
remoteL2-address	ULLA_STRING_t	Remote link layer address (L2 address): Unicast link: L2 address of the remote peer Broadcast link: L2 broadcast address Multicast link: L2 address of the multicast group Examples:_"01:02:03:04:05:06\0", IMEI = 16 bytes, etc. The maximum size of the null-terminated character array is 64 bytes.
rxBitRate	ULLA_INT_t	Downlink bitrate in bits per second
rxEncryption	ULLA_INT_t	Indicate whether downlink encryption is supported. To be interpreted as a boolean.
rxJitter	ULLA_INT_t	Link jitter in the receive path given in microseconds.
rxLatency	ULLA_INT_t	Link latency in the receive path
rxNoise	ULLA_INT_t	Received interference and noise. Usual devices cannot differentiate between noise and interference so that this attribute covers both values.
rxQuality	ULLA_INT_t	Quality of the link in the receive path given in percentage [0..100]. The algorithm how to calculate this value is LLA-implementation dependent. Possible approaches might incorporate the received signal strength, error rates, etc.
rxSignalStrength	ULLA_INT_t	Signal strength of the received signal given

Attribute Name	Type	Description
		in dBm.
signature	ULLA_STRING_t	Signature computed with hash function (16 bytes). Null terminated character array.
simultaneous	ULLA_INT_t	This denotes the number of ullalinks that can be used in parallel from the set of this link and those listed in the coexistsWith attribute.
state	MediaState_t	State of the link (read-only).
txBitRate	ULLA_INT_t	Uplink bitrate in bits per second
txEncryption	ULLA_INT_t	Indicate whether uplink encryption is supported. To be interpreted as a boolean.
txJitter	ULLA_INT_t	Link jitter in the transmit path given in microseconds.
txLatency	ULLA_INT_t	Link latency in the transmit path
txMTU	ULLA_INT_t	Uplink Maximum Transfer Unit in bytes
txQuality	ULLA_INT_t	Quality of the link in the transmit path given in percentage [0..100]. The algorithm how to calculate this value is LLA-implementation dependent. Possible approaches might incorporate the transmit power, error rates, etc.
txSignalPower	ULLA_INT_t	Power used for transmitted signals given in dBm.
type	MediaType_t	Type of communication technology used, Examples: Bluetooth, IEEE 802.11, etc.

Operations

Method	Notes	Parameters
accept() void	When a connection request was received the link state was changed to PendingAuthentication. This command accepts the connection request and the link state is set to CONNECTED.	
close() void	Switch a link back to the DISCONNECTED state when it was listening before.	
connect() void	Connect the link given in the CmdDescr_t given as additional parameter.	
deleteLink() void	The link specified is deleted. This should usually only be used with aggregate links that were created before but might also be applied to other links.	
disconnect() void	Disconnect the link given in the CmdDescr_t given as additional parameter.	

Method	Notes	Parameters
listen() void	Switch a link to the LISTENING state.	
reject() void	When a connection request was received the link state was changed to PendingAuthentication. This command rejects the connection request and the link state is set to DISCONNECTING.	

D.2 UllaLinkProvider class

Attributes

Attribute	Type	Notes
coexistsWith	ULLA_INT_t	This is a list of LPs (i.e. IDs) that are allowed to coexist with this LP, i.e. they can operate in parallel with no problems.
dependentOn	ULLA_INT_t	This is a list of LPs (i.e. IDs) that this Link Provider has some sort of dependency on. This attribute simply provides an indication that performance problems can arise when using this LP with others the list. It does not attempt to quantify the possible interference levels.
excludes	ULLA_INT_t	This is a list of LPs (i.e. IDs) that are absolutely excluded from use when this LP is in use.
lpId	Id_t	Unique link provider identifier.
maxPowerConsumption	ULLA_INT_t	Maximum power consumption while continuously blasting at full power given in microwatts. This attribute should usually be constant but this depends on technology. INFORMATIONAL.
minPowerConsumption	ULLA_INT_t	Minimum power consumption in stand-by mode with no active communication given in microwatts. This attribute should usually be constant but this depends on the technology. INFORMATIONAL.
networkScanPeriod	ULLA_INT_t	Period of network scan used during link discovery given in milliseconds.
powerMode	PowerModes_t	Which power mode is currently used, e.g. power saving, sleep, etc.
simultaneous	ULLA_INT_t	This denotes the number of LPs that can be used in parallel from the set of this link and

Attribute	Type	Notes
		those listed in the coexistsWith attribute.
supplier	ULLA_STRING_t	Supplier of the software implementation. Null-terminated string.
title	ULLA_STRING_t	LP name
type	MediaType_t	Type of communication system, e.g IEEE 802.11. As LPs might support multiple technologies this attribute is a multi-value attribute.
version	ULLA_STRING_t	Null-terminated string describing the version.

Operations

Method	Notes	Parameters
createLink() void	The LP should create a new aggregate link.	
scanAvailableLinks() void	Start the scanning procedure for any available links in the surrounding. Used to enable a forced scan for available networks.	