Körtner & Muth GmbH

LbwELI – Lockbase Electronic Lock Interface

Rev. 0.3.1846

Introduction

The LOCKBASE electronic lock interface (ELI) was designed to operate a wide range of electronic and mechatronic locks of different manufacturers by the same user interface.

Manufacturers of electronic locks usually are focused on the development and improvement of the technical capabilities of their devices. They also provide software, but this component often is not the best part of the product.

End users of electronic lock systems often administrate not only electronic, but also mechanic lock systems and often they administrate not only systems of the same manufacturer. It is in the interest of the end user to use the same software for mechanic lock systems as well as for electronic lock systems of different manufacturers.

For this reason LOCKBASE ELI consists of two components: (a) The common user interface, which is the same for all supported products, and (b) the manufacturer or even product specific 'driver' component, which operates the hardware according to the input received from the user interface module.

This specification describes the interface used for the communication between the 'driver' and the user interface module. It allows manufacturers of electronic and mechatronic lock systems to provide just a 'LOCKBASE ELI driver' with their products, instead of creating and maintaining a complete application program for end user administration, including user interface, help system, language support and all the other components required by a complete application program.

Basic Grammar and Conventions

The interface data will be exchanged in form of C-strings, sequences of 8-bit-characters, terminated by a null character (ASCII 0x00). Character encoding is UTF-8.

The syntax for a variable definition is “[variable]=[value]”. Each variable definition begins at the first position of a text line and will be terminated by a new line character (ASCII 0x0A). Each variable definition must appear only once.

The variable name uses alphanumeric characters (a-z, A-Z, 0-9) only. The equal sign may be preceded and followed by space characters, which will be ignored.

The value begins at the first non-space character at the right side of the equal sign and reaches until the last non-space character before the line feed. Value components containing newline characters must be quoted.

In this specification value expressions will be written in the form “[type]:[name]”, enclosed in square brackets, e.g.:

```
[ID:short name]
```

In the interface data stream the values must not contain type definitions, of course.

For values the following data types are defined:

B64	Base64 encoded binary data stream (RFC 4648, 'base64').
DEC	Numerical value in decimal notation and the dot ('.') as decimal separator.
DT	Date-Time value in the form

[YYYYMMDD]'T'[hhmmss]['Z'|''],

where YYYY=year with century, MM=month (1-12), DD=day of month (1-31), hh=hour (0-23), mm=minute (0-59) and ss=second (0-59). Date and time component are separated by a capital 'T', the time component is followed by an optional capital 'Z', which stands for UTC (absolute) time. Without the trailing 'Z' it is local (floating) time (see RFC 5545, 'DATE-TIME' value). Example:

```
20120508T105000Z
```

is the 8th of May, 2012 at 12:50:00 (CEST).

ID Identifier, uses alphanumeric characters only, e.g.:

```
KMGmbH
```

INT Integer in decimal notation

LTL Localized Text List, CSV line (comma separated, double quotation marks) of text phrases, each preceded by an ISO 639-1 language identifier followed by a colon e.g.:

```
“de:Deutsch”, ”en:English, british”, “fr:Français”
```

A text phrase with an empty (=starting with a colon) or without a language identifier (=without a colon at all) is regarded to be the default localization and will be used for all languages not explicitly listed. Only one default text localization is allowed per list.

RR Recurrence rule for a time period definition in the form

[INT:Multiplier][Time Interval]'('[List of values]')',

where '[Time Interval]' is one of the identifiers 's'=second (range 0-59), 'm'=minute (range 0-59), 'h'=hour (range 0-23), 'DW'=day of week (range Mo,Tu,We,Th,Fr,Sa,Su), 'DM'=day of month (range 1-31), 'DY'=day of year (range 1-366), 'DWM'=day of week per month (range Mo1-Su5), 'DWY'=day of week per year (range Mo1-Su53), 'W'=week of year (range 1-52), 'M'=month (range 1-12) or 'Y'=year (range is Gregorian year numbers). The leading multiplier and the time interval define the recurrence interval (e.g. '3M'=every third month, '2WY'=every second week). If not given, the leading multiplier is '1'. The recurrence interval is followed by an (optional) list of values to reduce the recurrence to certain values of the possible range (e.g. 'DW(Fr+Sa+Su)' means not every day, as 'DW' would, but only every Friday, Saturday and Sunday). The list is included in brackets, the values are separated by plus ('+') signs. Continuous sequences may be combined to 'from'-'to'-ranges (e.g. 'DW(Mo-We+Fr-Su)' would be the whole week except Thursday). Negative values may be used to indicate counting from end (e.g. 'DM(-1)' for the last day of a month).
Combined recurrence rules are separated by semicolon (';'), the resulting recurrence is the intersection of all listed recurrence rules. The first week is the first week with more than 3 days in a year.

Examples:

`DW(Fr)`	every Friday
`DWM(Mo1)`	first Monday of a month
`DWY(Su-1)`	last Sunday of a year
`DWM(Su1+Su3);M(1)`	every first and third Sunday in January
`WY(23+35)`	23rd and 35th week of the year
`Y(2012-2014)`	year 2012 to 2014

TL Text List, CSV record (comma separated, double quotation marks) of text phrases

TXT Text, must be quoted if containing any separator characters.

Interface Overview

ELICreate()	Constructor, the application identifies by licence number and revision and provides a callback function
ELIDestroy()	Interface destructor, the application detaches from the driver
ELIDriverInfo()	Application requests driver global information, e.g. revision and supported products
ELIDriverUI()	Application launches driver's user interface on user command, e.g. a driver provided dialogue box, etc.
ELIProductInfo()	Application requests information about capabilities and limitations of a certain product, e.g. programming capacity or event types
ELISystemInfo()	Application requests information about available systems depending on licence, revision and users
ELIOpen()	Application logs into a system and receives a session id
ELIApp2Drv()	Application launches a job in a given session
ELIDrv2App()	Driver's response to an application's request
ELIClose()	Application closes a given session (pending jobs will still be processed)

Construction and destruction

```
const char* ELICreate(    const char* sLic, const char* sLbwELIRev,
                          (*ELIDrv2App)() );
```

ELICreate() is the constructor of the interface object, which is required to use the interface. It expects the application to identify itself by its licence and LbwELI revision number, which allows the driver to support different interface revisions and adjust its services to different types of installations (e.g. factory, locksmith or end user installation). Second, the constructor expects a callback function pointer to launch the driver's responses to job requests (s.b.).

In case the constructor supports the requested interface revision, it simply returns EOK. In case it does not, it returns EREV, followed by the best interface it supports:

```
[ID:Error],[TXT:DrvELIRev]
```

This allows the application to 'downgrade' to an older drivers capabilities. If the driver cannot start for any other reason, the constructor returns EUNKNOWN.

```
void ELIDestroy();
```

ELIDestroy() is the destructor of the interface object, it will be called on application termination or when the application detaches from the driver for other reasons. The application will call the destructor anyway, even if the constructor did not returned EOK. Moreover, in case of revision problems, the application may call ELICreate() - ELIDestroy() several times.

Information Functions

Driver Information

To retrieve global information about the driver the application will call the interface function ELIDriverInfo():

```
const char* ELIDriverInfo();
```

The function returns a text pointer containing definitions for the following variables (s.b.). The returned text must be available for the application at least until the next call of an interface function. The information returned by ELIDriverInfo() is expected to be unchanged within the same driver revision.

The following variable definitions of the driver info are required:

```
DriverRevision =          [DEC:driver revision]
```

The driver's revision as numerical value ([Major].[Minor]). Required.

```
LbwELIRevision =          [DEC:supported LbwELI revision]
```

The supported interface revision as numerical value ([Major].[Minor]). Required.

```
Manufacturer =            [ID:short name],[LTL:company name]
```

The manufacturer of the device identified by an ID, the language independent manufacturer's short name, and a localized text list with more verbose variants of the company's name in different languages. Required.

```
Products =                [ID:product 1],[ID: product 2],...
```

The list of products supported by this driver, separated by comma, each identified by its short name (ID). Product ids will be used by the application to identify the product of new systems and to retrieve detailed information about the capabilities of supported products (s.b., ELIProductInfo()).

The following variable definitions of the driver info are optional:

```
DriverAuthor =            [LTL:driver's author]
```

Any information about the driver's author as localized text list. Optional.

```
DriverCopyright =         [LTL:copyright]
```

Copyright information about the driver as localized text list. Optional.

```
DriverUI =                [LTL:menu command]
```

Defines the title of a menu command to launch the driver's own user interface as localized text list. If this variable was defined, the application will create a menu entry for the driver and run the interface function ELIDriverUI() in case the menu was selected, i.e.

```
void ELIDriverUI( const char* SessID );
```

ELIDriverUI() will receive the application's current session id as a parameter (s.b.). This variable is optional, by default no driver provided user interface will be supported.

Product Capabilities and Limitations

The device capabilities and limitations are regarded to be product dependent. They will be requested by a call to the ELIProductInfo() method:

```
const char*  ELIProductInfo( const char* sProductID );
```

The function returns a text pointer containing definitions for the following variables (s.b.). The returned text must be available for the application at least until the next call of an interface function. The information returned by ELIProductInfo() is expected to be unchanged within the same driver revision.

The following variable definitions of the device capabilities are required:

```
Products =                [LTL:product name]
```

The human readable name of the product as localized text list. This variable is required.

```
ProgrammingTarget =       ['0':by keys|'1':by locks]
```

Defines whether keys or lock devices carry the access information and, therefore, programming should be performed by keys or by locks. This variable is required.

```
DeviceCapacity =          [INT:Maximum keys per lock resp. locks per key]
```

Defines the maximum number of key medium ids to be programmed into a lock device resp. lock device ids to be programmed into a key medium, the device capacity. This variable is required.

```
TimePeriodCapacity =      [INT:Maximum time periods per device]
```

Defines the maximum number of time periods to be programmed into a device (either key medium or lock device), the time period capacity. This variable is required. A value of '0' indicates the system is not capable of time dependent programming at all (e.g. just by number of lockings, etc.).

```
EventTypes =              [ID:EventID0],[ID:Class],[LTL:Description0];
                          [ID:EventID1],[ID:Class],[LTL:Description1];...
```

The list of event types the product supports, separated by semicolon, each defined by its unique ID, its class and a localised event description. The class must be one of the predefined values 'security' (unauthorized key access), 'technical' (battery low, power off, maintenance, etc.), 'access' (authorized key access) or 'other' (defined by device). The event description is made up of an event name and an event description, separated by a colon, e.g. “en:Power off:The device was powered off” (note that in this case a default entry in the localized text list MUST begin with a colon).

The following variable definitions of the device capabilities are optional:

```
OnlineSystem =            ['0':offline|'1':online]
```

Programming modus of the system, either 'online' (synchronous data exchange) or 'offline' (asynchronous data exchange). This variable is optional, the default is '0' (offline, asynchronous data exchange).

```
DefaultAccess =           ['0':never|'1':always]
```

The default access policy for devices with no access definition at all, possible values are '0' (closed by default) and '1' (open by default). This variable is optional, the default is '0' (closed by default).

```
AccessByNmbOfLockings =   ['0':no|'1':yes]
```

The device allows to program a certain (limited) number of lockings for a key medium, possible values are '0' (no) and '1' (yes). This variable is optional, the default is '0' (no).

```
AccessByFloatingPeriod=   ['0':no|'1':yes]
```

The device allows to program just an access duration (e.g. 24 hours since first access), possible values are '0' (no) and '1' (yes). This variable is optional, the default is '0' (no).

```
TimePeriodRecurrence =    [ID:RecIntID1],[ID:RecIntID2],...
```

Defines the recurrence interval types supported by the product in a comma separated list. 'RecIntID' is one of the values 's', 'm', 'h', 'DW', 'DWM', DWY', 'DM', 'DY', 'W', 'M' or 'Y', as defined in the RR data type. This variable is optional, by default no recurrence is supported at all.

```
EventUpdateInterval =     [INT:Seconds]
```

Defines the minimal interval for event polling requests by the application in online systems. The default is 60 seconds, however, it will be ignored for offline systems.

```
AccessUpdateInterval =    [INT:Seconds]
```

Defines the minimal interval for the re-programming of the access rights by the application in online systems. The default is 60 seconds, however, it will be ignored for offline systems.

System Information

To retrieve information about the available systems and the access rights the application has for a given system, the application calls the ELISystemInfo() function:

```
const char* ELISystemInfo( const char* sUsers );
```

The function expects the current logged user(s). If multiple users are logged in, the 'sUsers' parameter is a CSV list (TL) of the logged users. Together with the applications licence information and revision the driver can compute the access rights the user(s) may have for each available system.

The function returns a list of CSV records, which contains a line for each available system. Additionally for each product the driver supports, it includes a line listing the access rights the application would have for newly created systems (if the application is not allowed to create new systems at all, these lines may be omitted):

```
[ID:Sys1],[ID:ProductID],[TXT:Name],[ACLR],['0':disable|'1':enable]
[ID:Sys2],[ID:ProductID],[TXT:Name],[ACLR],['0':disable|'1':enable]
...
,[ID:Product1],,[ACLR]
,[ID:Product2],,[ACLR]
...
```

Each line gives the id of the system, its product id, an (optional) human readable name, the access rights the application would have in this system and the systems current state (enable or disable). The access rights are a combination of the capital letters 'A' for 'Access programming', 'C' for 'Create objects/systems', 'L' for 'List systems' and 'R' for 'Remove objects/system'.

The lines for new systems are of the same form as the lines for available systems, but it contain only product id and access rights.

Open and Close

Open/Create

```
const char* ELIOpen(      const char* sUserList,
                          const char* sSystem = NULL,
                          const char* sExtData = NULL,
                          const char* sContinue = NULL );
```

The ELIOpen() function opens a known system or creates a new one. The function expects the current user(s) and the id of the system to open. In case the driver created a binary data block to store with the system in the application's database (s.b., ELIClose()), this extra data bIock will be passed via 'sExtData' (base64 encoded). The optional parameter 'sContinue' may contain a list of sessions with outstanding job responses, indicating that this new session should be used to pass them to the application (s.b.).

If the 'sSystem' parameter is NULL, the application requests the creation of a new system. In this case the 'sExtData' parameter must contain the product id of the new system.

The ELIOpen() function returns a CSV record of the following form:

```
[ID:Error],[ID:SessID],[ACLR],[ID:LastSessID],['0':disable|'1':enable]
```

In case the function succeeds, the error field contains 'OK' and is followed by the session id, the access rights, the last recent session id and the system state (s.a.). In case an error occurs the error field contains one of the defined error codes (see 'Data Exchange, Jobs and Statements, Error Codes') and the following fields are omitted.

The session id connects the opened session to the access control system given in the 'sSystem' parameter. It will be used in subsequent calls of ELIApp2Drv() to refer to the opened system. The last recent session id is the last session id the driver processed for this system. It is useful for the application to check, whether driver and application state of the system are identical. Old session ids may also be used by the application to continue a session with outstanding job responses in ELIOpen().

The driver must assure, that only a single session exists for a certain system at the same time. The session id must be unique among all sessions currently open and all sessions with outstanding job responses (s.b.).

Close

```
const char* ELIClose( const char* sSessID );
```

By using the Close() function the application signals the end of a session to the driver. The function expects the session id returned by a previous call of ELIOpen().

The ELIClose() function returns a CSV record of the following form:

```
[ID:Error],[B64:ExtData]
```

In case the function succeeds, the error field contains 'OK' and is (optionally) followed by a binary data block to store with the system in the application's database. In case an error occurs the error field contains one of the defined error codes (see 'Data Exchange, Jobs and Statements, Error Codes') and the following fields are omitted.

Data Exchange

Jobs and Statements

Communication and data exchange between driver and application is done by job requests sent by the application and appropriate job responses sent by the driver. These two functions are used to send job requests and job responses:

```
int ELIApp2Drv( const char* sSessID, const char* sJobID, const char* sJobData );
int ELIDrv2App( const char* sSessID, const char* sJobID, const char* sJobData );
```

Each function expects a session id, a job id and the job's data. The job id is a unique job identifier created by the application, the data block contains the job's data, command and return statements. Both functions return a value of zero on success or a value unequal zero in case of an error.

The application must assure the job identifier is unique within the scope of the system as long as outstanding job responses exist.

A job can be send or returned in more than one step. As long as session id and job id are the same, subsequent calls to ELIApp2Drv() or ELIDrv2App() refer to the same job. The job may be split over several function calls to allow seamless processing of even huge jobs. In case a job data block is incomplete it must be terminated by a JC statement (s.b.) to indicate the data transfer for this job will be continued.

In the examples of this specification the function calls will be shown in an abstract form by a leading 'APP2DRV/DRV2APP [SessID [JobID] BEGIN' line, followed by the job data and a trailing 'END' line. Please note that the 'headers' and 'footers' in the examples do not belong to the order data, but only serve to symbolize the function call.

The job's data block is a list of statements. A statement is a CSV line representing a command, object or return value. It starts with a statement identifier and is followed by a coma separated list of parameters. The meaning of the parameters depends on the statement.

Statement Overview

The following statements are defined (in alphabetic order):

AK, AKR	Key medium access policy statement and assorted return statement
AL, ALR	Lock device access policy statement and assorted return statement
AT, ATR	Access table statement and assorted return statement
CK, CKR	Key medium create statement and assorted return statement
CL, CLR	Lock device create statement and assorted return statement
DK, DKR	Key medium data statement and assorted return statement
DL, DLR	Lock device data statement and assorted return statement
EK	Key medium event statement
EL	Lock device event statement
JC	Job continue statement
LD, LDR	Data list statement and assorted return statement
LE, LER	Event list statement and assorted return statement

RK,RKR	Remove key medium and assorted return statement
RL, RLR	Remove lock device and assorted return statement
RS, RSR	Remove system and assorted return statement

Error Codes

The following error codes are used by the information and data exchange functions:

OK	Ok, no error
EREV	Invalid/ unsupported revision
ESESS	Invalid session id
EDISABLE	System temporarily not available (offline)
EACCESS	Access to system or object denied
EPRODUCT	Creation of a new system fails due to wrong product id
ECAPACITY	Request fails due to capacity problems
EBUSY	Request fails because driver is busy (too many requests per time unit)
EINVNUM	The number of access definitions of an access table definition is invalid
EINVPERIOD	A time period definition in an access table definition is invalid
EINVSTAT	The statement is invalid, e.g. one or more of the parameters are invalid or missing
EKEY	The physical key medium does not work properly or returns an error (damaged, out of order, etc.)
ELOCK	The physical lock device does not work properly or returns an error (damaged, out of order, etc.)
EINVDK	A required DK statement is missing or invalid
EINVDL	A required DL statement is missing or invalid
EINVAT	A required AT statement is missing or invalid
EUNKNOWN	An unknown problem occurred, which cannot be analysed and requires a restart of the driver

Object data statements

```
DK,[ID:KID],[TXT:Name],[ID:AppID],[B64:ExtData]
DL,[ID:LID],[TXT:Name],[ID:AppID],[B64:ExtData]
```

Object data statements are used to interchange information about object states between driver and application and to link driver ID and corresponding application ID (e.g. on create and list statements).

Defined data statements are DK for key media and DL for lock devices. Their parameters are the driver's object id, a human readable name, the application's object id and a binary data block defined by the driver, but stored in the application's database (base64 encoded, optional). Depending on purpose and context, each single parameter may be omitted. The binary data

block will be modified only by the driver. If defined, the application will present it in a job request and update it from a job response.

The DKR resp. DLR statement will be used by the driver to indicate problems with a requested object by returning one of the error codes EINVDK, EINVDL, e.g.:

```
DLR,[ID:LID],EINVDL
```

Access table statement

```
AT,[ID:TID],[INT:MinNmbOfLockings]-[INT:MaxNmbOfLockings],
      [!][DT:Start0]/[INT:Duration0]/[RR:Recurrence0]/[DT:End0],
      [!][DT:Start1]/[INT:Duration1]/[RR:Recurrence1]/[DT:End1],...
```

The access table statement defines an access policy for key media in lock devices. Parameters are a table identifier, an (optional) number of lockings definition and a list of time period definitions.

The number of lockings definition is made up of a minimum and a maximum value, separated by a dash ('-'). Minimum and maximum are minimum resp. maximum number of successful accesses allowed, regardless of time period definitions. The minimum must be less than the maximum. If only a single value is given, it is regarded to be the minimum value. If no value is given, no limitation by number of lockings is defined.

The time period definition has four parts: Start time, duration, recurrence rule and end time. Not only recurrence rule and end time, also start time and duration may be omitted. No start time means 'start at first usage', no duration means 'unlimited'. No time period definition at all means 'just the given number of lockings, regardless of time'. Recurrence rule and end time are valid only if start time and duration are defined. The end time is optional and ends the recurrence, which would be unlimited otherwise.

A recurrence rule refers to the start time (a mismatch of start time and recurrence rules leads to an EINVPERIOD error) and makes the time intervals defined in the rule and all higher level intervals a variable position in the start time. E.g. if a monthly recurrence was defined, month and year in the start time become variable (according to the recurrence rule definition), while day, hour, minute and second remain fixed as defined in the start time. A procedure to expand a recurrence rule would use start time and duration to complete the derived events to fully defined time periods.

An exclamation mark (!) as first character of a time period definition may be given to negate the time period defined afterwards. This may be useful to define exceptions from a recurring event.

Leap seconds will not be taken into account when calculating the end date/time of a time period from start date/time and duration (a day is assumed to last always 86400 seconds).

The ATR statement will be used by the driver to indicate problems with the access table definition by returning one of the error codes OK, ESESS, EDISABLE, EACCESS, ECAPACITY, EINVNUM, EINVPERIOD, e.g.:

```
ATR,[ID:TID],EINVPERIOD
```

Access policy statements

```
AK,[ID:KID],[ID:TID],[ID:LID1],[ID:LID2],[ID:LID3],...
AL,[ID:LID],[ID:TID],[ID:KID1],[ID:KID2],[ID:KID3],...
```

The access policy statement links access tables to pairs of key media and lock devices. The parameters of an access policy statement for keys (AK) are a key, an access table and a list of lock devices. Thus the access policy defined in the access table will be assigned to the relation of

the given key medium and each of the listed lock devices. Accordingly an access policy statement for locks (AL) has a lock, an access table and a list of key media. Whether AK or AL statements are to use depends on whether keys or lock devices hold the access information.

The return will be an AKR resp. ALR statement with one of the error codes OK, ESESS, EDISABLE, EACCESS, EINVSTAT, EKEY, ELOCK, EINVDK, EINVDL or EINVAT, e.g.:

```
AKR,[ID:KID],OK
ALR,[ID:LID],EINVSTAT
```

List statement

The list statements allow the application to read information of already existing systems for further administration, to resynchronize in case the current state of the application's database seems to be invalid or out of date and to request the processing state of scheduled jobs.

```
LD
```

The driver will return a DK statement for each key media, a DL statement for each lock device, an AT statement for each access table and AK or AL statements for each key medium resp. lock device holding access information. Whether AK or AL statements will be returned depends on whether keys or lock devices hold the access information.

Example:

```
ELIApp2Drv [SessID] [JobID] BEGIN
LD
END
```

Driver's response:

```
DRV2APP [SessID] [JobID] BEGIN
DK,[ID:KID1],[TXT:Name1],,0,[B64:ExtData]
DK,[ID:KID2],[TXT:Name2],,0,[B64:ExtData]
...
DL,[ID:LID1],[TXT:Name1],,0,[B64:ExtData]
DL,[ID:LID2],[TXT:Name2],,0,[B64:ExtData]
...
AT,[ID:TID1],0-5,20120508T105000Z/153040,20120608T105000Z/173040
AT,[ID:TID2],,20120509T220000Z/3600,20120510T220000Z/3600
...
AK,[ID:KID1],[ID:TID1],[LID1],[LID2],[LID3],...
AK,[ID:KID1],[ID:TID2],[LID3],[LID5],[LID7],...
AK,[ID:KID2],[ID:TID1],[LID1],[LID3],[LID4],...
...
LDR,OK
END
```

The driver will return a terminating LDR statement with one of the error codes OK, ESESS, EDISABLE, EACCESS.

Create new objects

The application can create new key media or lock devices by sending a CK or CL statement.

Example:

```
APP2DRV [SessID] [JobID] BEGIN
CK,[ID:AppID1],[TXT:Name]
CK,[ID:AppID2],[TXT:Name]
...
CL,[ID:AppID1],[TXT:Name]
CL,[ID:AppID2],[TXT:Name]
```

```
...
END
```

Driver's response:

```
DRV2APP [SessID] [JobID] BEGIN
DK,[ID:KID1],[TXT:Name],[ID:AppID1],[B64:ExtData]
DK,[ID:KID2],[TXT:Name],[ID:AppID2],[B64:ExtData]
...
DL,[ID:LID1],[TXT:Name],[ID:AppID1],[B64:ExtData]
DL,[ID:LID2],[TXT:Name],[ID:AppID2],[B64:ExtData]
...
END
```

CK and CL statements contain an application provided id for each object to create. The driver replies with a DK or DL statement for each requested object and uses the given application id to link its own electronic ID to the application's object.

In case of an error the driver replies with a CKR or CLR statement, which contains the application id of the requested object and one of the error codes ESESS, EDISABLE, EACCESS, EINVSTAT, ECAPACITY, e.g.:

```
CKR,[ID:AppID1],ECAPACITY
```

Remove objects

The application can remove existing key media or lock devices from a system by sending RK or RL statements.

Example:

```
APP2DRV [SessID] [JobID] BEGIN
RK,[ID:KID1],[B64:ExtData]
RK,[ID:KID2],[B64:ExtData]
...
RL,[ID:LID1],[B64:ExtData]
RL,[ID:LID2],[B64:ExtData]
...
END
```

Driver's response:

```
DRV2APP [SessID] [JobID] BEGIN
RKR,[ID:KID1],OK
RKR,[ID:KID2],OK
...
RLR,[ID:LID1],EINVSTAT
RLR,[ID:LID2],OK
...
END
```

The driver replies with RKR resp. RLR statements, which contain the id of the target object and one of the error codes ESESS, EDISABLE, EACCESS, EINVSTAT.

Access programming

Access programming will be done by a list of AK resp. AL statements in a job. Because AK/AL statements require AT statements to be resolved, a list of all required AT statements will precede the AK/AL statements. And in case binary data blocks had been defined for key medium resp. lock device objects, also DK and DL statements for each mentioned object will precede the AK/AL statements. Thus an access programming job always contains all the required information to process the job.

Example:

```
APP2DRV [SessID] [JobID] BEGIN
DK,[ID:KID1],,[B64:ExtData]
DK,[ID:KID2],,[B64:ExtData]
...
DL,[ID:LID1],,[B64:ExtData]
DL,[ID:LID2],,[B64:ExtData]
...
AT,[ID:TID1],0-5,20120508T105000Z/153040,20120608T105000Z/173040
AT,[ID:TID2],,20120509T220000Z/3600,20120510T220000Z/3600
...
AK,[ID:KID1],[ID:TID1],[LID1],[LID2],[LID3]
AK,[ID:KID2],[ID:TID2],[LID1],[LID3],[LID4]
END
```

Driver's response:

```
DRV2APP [SessID] [JobID] BEGIN
DK,[ID:KID1],,[B64:ExtData]
DK,[ID:KID2],,[B64:ExtData]
...
DL,[ID:LID1],,[B64:ExtData]
DL,[ID:LID2],,[B64:ExtData]
...
ATR,[ID:TID1],OK
ATR,[ID:TID2],OK
...
AKR,[ID:KID1],EKEY
AKR,[ID:KID2,OK
...
END
```

The return will contain an ATR statement for each AT statement in the request and an AKR resp. ALR statement for each AK resp. AL statement. In case the driver needs to update the binary data block stored in the applications database with each object, it will also include DK and DL statements with updated binary data blocks in its response.

Asynchronous Programming

Depending on the technology and capabilities of the access control system, it might happen, that the application has to close the session before the driver was able to respond to all job requests within a session.

In this case the driver will store the pending job responses and wait until the application opens a new session with a continue request for the terminated session (s.a., parameter 'sContinue' of the ELIOpen() function). The driver then will pass the pending job responses to this new session instead of the terminated one.

Event polling

```
LE,[DT:Since]
```

Event polling will be done by the special list statement 'LE' with an (optional) date time parameter to define the time since which events should be returned. The driver's return is a list of event statements:

```
EK,[ID:KEID],[ID:EventType],[DT:Time],[INT:Milliseconds],[ID:LEID]
EL,[ID:LEID],[ID:EventType],[DT:Time],[INT:Milliseconds],[ID:KEID]
```

There are key medium (EK) and lock device (EL) related events. Its parameters are the id of the belonging object (key or lock), the type of the event (one of the event types listed in the

'EventTypes' value returned by the ProductInfo() function), the time of the event's occurrence, an (optional) millisecond of the event time and an (optional) second object id, which depends on the type of the event (e.g. the used key on access events of a lock). The second object id is required for all events signalling an interaction between two 'known' objects.

Example:

```
APP2DRV [SessID] [JobID] BEGIN
LE,[DT:Since]
END
```

Driver's response:

```
DRV2APP [SessID] [JobID] BEGIN
EL,[ID:LEID1],[ID:EventType],[DT:Time],[INT:Milliseconds],[ID:KEID1]
EL,[ID:LEID1],[ID:EventType],[DT:Time],[INT:Milliseconds],[ID:KEID2]
...
EK,[ID:KEID1],[ID:EventType],[DT:Time],[INT:Milliseconds],[ID:LEID1]
EK,[ID:KEID2],[ID:EventType],[DT:Time],[INT:Milliseconds],[ID:LEID1]
...
LER,OK
END
```

In the driver's response the list of events will be terminated by a LER statement with one of the error codes OK, ESESS, EDISABLE, EACCESS or EBUSY. The driver will return EBUSY with online systems in case the event polling interval of the application is too short (i.e. if there is still a pending LE statement). If the driver needs to update the binary data block of the system record, it may include a resp. DS statement in its response.

Implementation Notes

The driver is to compile with 32 bit address size. Its implementation may be single threaded. By definition only a single application thread will use the interface. An application may use several drivers at a time, but a certain driver will be operated by just one application thread.

Data will be transferred over the interface by passing or returning pointers to the sender's internal storage locations. This storage is 'read only' for the recipient, it must not write to these locations. The recipient must copy the transmitted data immediately into its own storage. Pointers and data returned by a function must be guaranteed to be valid until the next call of an interface function by the recipient (or until the sender's termination).

The final binary of the driver (Dynamic Link Library, DLL in Windows) should be named according to the following convention to be recognized by LOCKBASE:

```
LbwELI.[Manufacturer].dll
```

Example:

```
LbwELI.Abus.dll
```

Change Log

0.3.1846	• DriverInfo() returns product ids only, product name will be returned by ProductInfo now.
0.2.1844	• New optional parameter 'sContinue' to pass to to ELIOpen() to request to continue sessions with outstanding response statements • JobID parameter changed from int to const char* and made unique

	beyond the session • sSysID parameter removed from ELIDriverUI() parameter list
0.1.1837	• Document revision number added • Function name prefix 'ELI' added • ELIDestroy() added • Drivers revision number added to return of 'ELICreate()'