Database Management System LINTER®

Version 5.9

Linter API

Relational Expert Systems

Table of contents

Introduction 5

Structure of LinAPI 6

Connections 6
Cursors 6
Statements 6

Basic Elements 7

Error Diagnostics 7
Synchronous and Asynchronous Query Execution 7
Stages of Query Processing 8
Connections 8
Cursors 9
Statements 9
Multithreading support 10
Database Objects 10
Data Types in LinAPI 10
Application link and compilation with LinAPI library 11

Standard Application Structure 12

Functions 13

Connections 13
  Open Connection 13
  Close Connection 14
  Open connect asynchronously 14
  Get Connection Attributes 14
  Set Connection Attributes 15
Cursors 16
  Open Cursor 16
  Close Cursor 16
  Get Cursor Attributes 17
  Set Cursor Attributes 17
Statements 18
  Create Statement 18
  Free Statement 19
  Get Statement Attribute 19
  Execute Statement 20
Processing Queries and Parameters 20
  Bind Query Parameter 20

Bind Answer Buffer....22
Unbind Answer Buffer....23

Transactions and Locks....24
Commit Changes on Cursor....24
Confirm Changes on Connection....24
Rollback Changes on Cursor....25
Rollback Changes on Connection....25
Lock Row....25
Unlock Row....26

LINTER_ExecuteDirect....26

LINTER_Fetch - cursor positioning....27

Get Answer Row to Buffer....28

Get Answer Column to Buffer....29

Working with BLOBs....29
Append Data to BLOB....30
Select portion of BLOB data....31
Clear BLOB data....32
Get or Set BLOB Type....32
Get Size of BLOB....33
Purge blob data for specified by number blob column....33
Add BLOB data to specified by number BLOB column....33
Retrieve portion of specified BLOB column....34

LinAPI Messages....35
Get Message Text....35
Get Linter Completion Code....36

Check Connection Operation Completed....37

Check Cursor Operation Completed....37

Get DB Object Description....38

End Session....39

Close database channel....39

Retrieve database server information....39

Cancel request processing....40

Programming Hints....41

Asynchronous Programming....41

Working with Stored Procedures....42
Processing Compilation Errors....42
Processing Returned Values and Output Parameters....42
When Returned Value is of the Cursor Type....43

LinAPI Completion Codes....44

Characteristics of LinAPI Objects....45

Connection Attributes....45

Cursor Attributes 45
Statement Attributes 45
LinAPI Data Types 46

Introduction

This document describes the syntax and usage of Linter's application interface library LinAPI. It is written for programmers working in CC++, Pascal (Delphi), and similar programming languages. The programming examples are in C

The LinAPI library is a set of functions supporting operation with the RELEX DB (Linter Series) with programs written in C, Pascal, or similar languages.

The library is distributed as lnapi.dll for MS Windows systems (linapi.a, linapimt.a for Linux) or as the object library lnapi.lb.

Subdirectories:

- LIB – export libraries for DLL;
- LIBVC32 – static libraries for MS Visual C compiler;
- LIBVC32MT – multithread static libraries for MS Visual C compiler;
- LIBWC32 – static libraries for Watcom C 10.5-11.x compiler.

For MS DOS and NetWare LinAPI is distributed with three options:

- linapib.lib – for Borland C, large memory mode;
- linapim.lib – for Microsoft C, large memory mode;
- linapiwf.lib – for Watcom C, flat memory mode.

Structure of LinAPI

LinAPI has three types of objects allowing to process database operations: Connections, Cursors, and Statements.

Each of these objects has a set of attributes, or characteristics, allowing to evaluate and change the state of the objects.

Connections

The LINTER_Connect function establishes a connection to a Linter server. When the connection is created, one channel to a Linter server is opened. After initializing a connection, the application is able to access a database that is currently active on the Linter server.

Any application may have multiple connections to one or more Linter servers concurrently.

Cursors

The LINTER_OpenCursor function opens a cursor for the specified connection. A cursor is an object used to process queries and statements. When a cursor is opened, a channel to a Linter server will be opened. This channel will be a child channel of the channel that opened his connection.

The cursor may be understood as a specific data area that describes the state of execution of a query or statement. Several statements may be connected to one cursor, but the cursor will contain the state of the query or statement last executed.

Statements

The LINTER_CreateStatement unction creates a statement that is the result of an SQL query compilation. Compilation is performed on the channel that was opened when the connection was created.

A query to be compiled may include parameters. Parameters should be treated as variables. To provide a statement with a specific parameter values, the parameter should be bound with the parameter buffer (which will contain the data to be used at the statement execution) to the statement.

It is reasonable to create a statement when there are an intension to use it many times or with parameters.

The same statement may be connected to multiple cursors where each cursor is associated with the connection on which the statement was created.

The cursor and statement are linked when the query parameters buffers and the result set column buffers are bound.

This link is passive, activated only when the statement is being executed.

Basic Elements

This section describes the basic elements and concepts of the LinAPI.

Error Diagnostics

All LinAPI functions return:

LINAPI_SUCCESS	if function completed successfully;
LINAPI_ERROR	in case of internal error;
LINAPI_INV_ID	if invalid identifier was transferred to a function;
LINAPI_NO_MEM	if there is not enough memory available;
LINAPI_BUSY	if any function called earlier for the same LinAPI object has not been completed yet;
LINAPI_INV_CONTEXT`	if LinAPI internal structures are damaged;
LINAPI_ID_NUM_EXCEEDED	if the number of used identifiers exceed limit.

On successful completion, all functions return zero, LINAPI_SUCCESS.

If the error code is LINAPIERROR, you have to call LINTER_Error function to get the Linter completion code.

Synchronous and Asynchronous Query Execution

If a function allows asynchronous processing, two parameters are required by LinAPI to be set:

AsyncFunc	pointer to user callback function;
UserArg	pointer to the user callback function parameter.

A function specified as AsyncFunc must have three parameters:

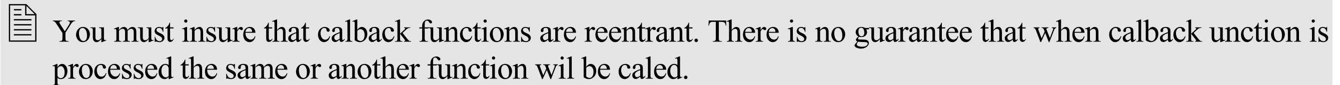
```
void UserAsyncFunc(
  short CursorID,      /* Cursor ID */
  long LastError,      /* Last LinAPI err /
  void *UserArg)       /* User defined parameter */
```

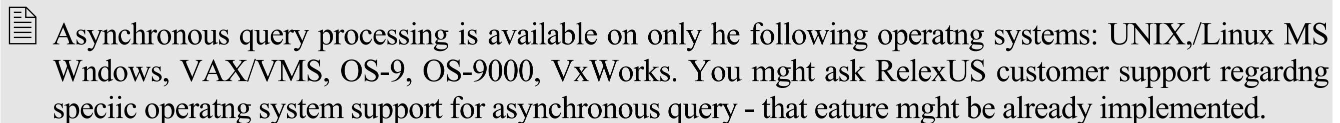
If both the AsyncFunc and UserArg pointers are NULL, the query is processed synchronously; i.e., program execution will resume only after query completion.

If both the AsyncFunc and UserArg pointers are not NULL, program execution continues without waiting for query processing completion, i.e., asynchronously. As soon as the query is completed, successfully or with an error, the application receives a query completion message. Program operation is interrupted and the specified user callback function is called (receiving UserArg). When AsyncFunc is completed, the program will resume execution from the interrupt point.

If the AsyncFunc pointer is NULL and the UserArg address is not N ULL, the query will be processed asynchronously without user callback unction call.

Using the LINTER_ConnectComplete or LINTER_CursorComplete functions, an application may determine if a query has been completed and, if it is not yet finished, enter a wait mode. These two functions are the only way to determine query completion.


You must insure that calback functions are reentrant. There is no guarantee that when calback unction is processed the same or another function wil be caled.


Asynchronous query processing is available on only he following operatng systems: UNIX,/Linux MS Wndows, VAX/VMS, OS-9, OS-9000, VxWorks. You mght ask RelexUS customer support regardng speciic operatng system support for asynchronous query - that eature mght be already implemented.

Stages of Query Processing

Each query in LinAPI may pass through several states:

1) Query compilation is useful for multiple executions. In this case, the query may contain unknown value parameters to be inserted into the query at the query execution stage.
2) Parameter binding.
3) Query execution.
4) Binding of answer fields to program variables from SELECT queries.
5) Fetching answers with an opportunity to move within a selected row set in SELECT queries.

Not all of these stages are necessary. If the query has no parameters and you do not plan to compile it, the irrelevant stages will be omitted. Binding answer fields may precede query execution.

Connections

Connection is an object intended for connecting to a DBMS. The connection should be created before opening cursors and creating statements. Besides, connection is useful for unification of several cursors into one transaction. Therefore, COMMIT/ROLLBACK command send to a connection affects all cursors of this connection. CLOSE command closes all the cursors of the specified connection.

The characteristics of connections are described in section “Connection Attributes”.

The cDBDesc connection characteristic provides information about a database in the following structure:

```
#define linNameLen 18
typedef struct {
   char SysLog;
   char BaseName[linNameLen];
   char Sync;
   char Log;
   char Os;
} t_DBDesc;
typedef struct {
   long VerMajor,            /* Linter version used to create DB */
        VerMinor,
        VerBuild;
   long SortPoolSize;        /* Size of sorting pool in pages */
   long KernelPoolSize;      /* Size of kernel pool in pages */
   long FileQueueSize;       /* Size of file queue */
   long UserQueueSize;       /* Size of user queue */
```

```
   long TableQueueSize;      /* Size of table queue */
   long ColumnQueueSize;     /* Size of column queue */
   long ChannelQueueSize;    /* Size of channel queue */
   long SnapTimeout;         /* Timeout between kernel cache Full Snap
                             operations (in minutes)*/
   long KillTimeout;         /* user process alive verification timeout */
   long LReserv1;
   long LReserv2;
   long LReserv3;
   long LReserv4;
   long LReserv5;
   Char                      /* Database name */
   BaseName[linNameLen];
   char SysLog;              /* Attribute of work with transaction log */
   char Sync;                /* Attribute of I/O synchronization */
   char Log;                 /* Attribute of log file presence */
   char Os;                  /* Identifier of server's operation system */
   char Pad[2];
} t_DBDesc;
```

Cursors

A cursor is an object used to execute queries and statements. You may send several queries in sequence on a cursor. A cursor is also a tool for organizing transactions. Several queries sent on a cursor may be completed using either the COMMIT command, which confirms all changes in the database, or ROLLBACK, which undoes pending changes.

Cursor attributes are described in section “Cursor Attributes”.

Statements

A statement is an object containing the result of a query compilation. Statement attributes are described in section “Statement Attributes”.

The sAnswerDesc statement attribute is a description of the returned column with the following structure:

```
#define     LinNameLen 66
typedef     struct    {
char        Owner[linNameLen];     /* User name */
char        Table[linNameLen];     /* Table name */
char        Column[linNameLen];    /* Column name */
Short       Length;                /* Type */
char        Type;                  /* Length */
char        Prec;                  /* Precision */
char        Scale;                 /* Scale */
char        NullIndicator;         /* NULL-value indicator */
long        RealLength;            /* Actual data length */
}  t_ParamDesc;
```

The query to be compiled may include parameters that will be processed at the bind stage.

The sParamDesc statement attribute is a description of a query or statement. It is also accepted in t_ParamDesc structure. If sParamDesc is included in t_ParamDesc, the Owner, Table, and RealLength fields are left blank.

Multithreading support

LinAPI providing full multithreading mode for applications: application threads can share LinAPI objects, such as connections, cursors, statements, transactions. LinAPI synchronizes the use of shared objects between threads.

Database Objects

The LinAPI will provide for returning information about various database objects: tables, columns, views, etc. The current version provides for only one object type, the view. In LinAPI, this type is denoted as oView.

Data Types in LinAPI

The data types used by the LinAPI library are:

Data Type	Description
tChar	array of one-byte characters
tNChar	two-bytes Unicode array
tVarChar	array of one-byte characters with identified number of valued data in array
tNVarChar	array of two-bytes Unicode characters with identified number of valued data in array
tByte	byte string
tVarByte	array of bytes with identified number of valued data in array
tString	NULL-terminated character string
tSmallint	16-bit integer
tInteger, tInt	32-bit integer
tBigInt	64-bit integer
tBoolean	8-bit integer
tReal	16-bit single-precision float decimal data
tDouble	32-bytes double-precision float decimal data
tNumeric, tDecimal, tDec	Precise numeric data with the fixed decimal point, equivalent to DECIMAL type in Linter
tDate, tTimeStamp	date and time; equivalent to DATE type in Linter
tBlob	BLOB data properties

Remarks

tBigInt data type operations are naturally supported on platforms with 64-bit integer support. For the platforms there the 64-bit operations are not defined, please use int64.a library. The functions definitions for int64.a library defined in int64.l1 file and RELEX DB distribution

contains the complete source code for the 64-bit operations in int64.c module. Use INT64_EMULATION_ENABLE definition as a compiler option.

tNumeric, tDecimal, tDec data type operations provided by decimals library. The size of that data type is 16 bytes. Please, refer to the correspondent RELEX DB documentation. The source code of decimals library is included into RELEX DB distribution.

tDate, tTimestamp data type operations provided by tick library. The size of that data type is 16 bytes. The source code of tick library is included into RELEX DB distribution. Please, refer to the correspondent RELEX DB documentation.

tExtFile data type is character buffer with the length of 255 bytes.

Application link and compilation with LinAPI library

The target executable application module should be linked with linapi library.

The compiler definitions for the

Operating System	Compilier definitions
Wn32	-DINTER_MSWINDOWS
	-DWIN32
	- D_VER_MAX=$(VERSION)
WnCE	
Linux	-DLINUX D_VER_MAX=$(VERSION)
Irix	-DIRIX
	-D_VER_MAX=$(VERSION)
Solaris	-DSOLARIS D_VER_MAX=$(VERSION)
VxWorks	-DXWORKS D_VER_MAX=$(VERSION)

The VERSION definition should be 590 for RELEX DB V.5.QX and 6ooforRelXDBv.6.o.x.

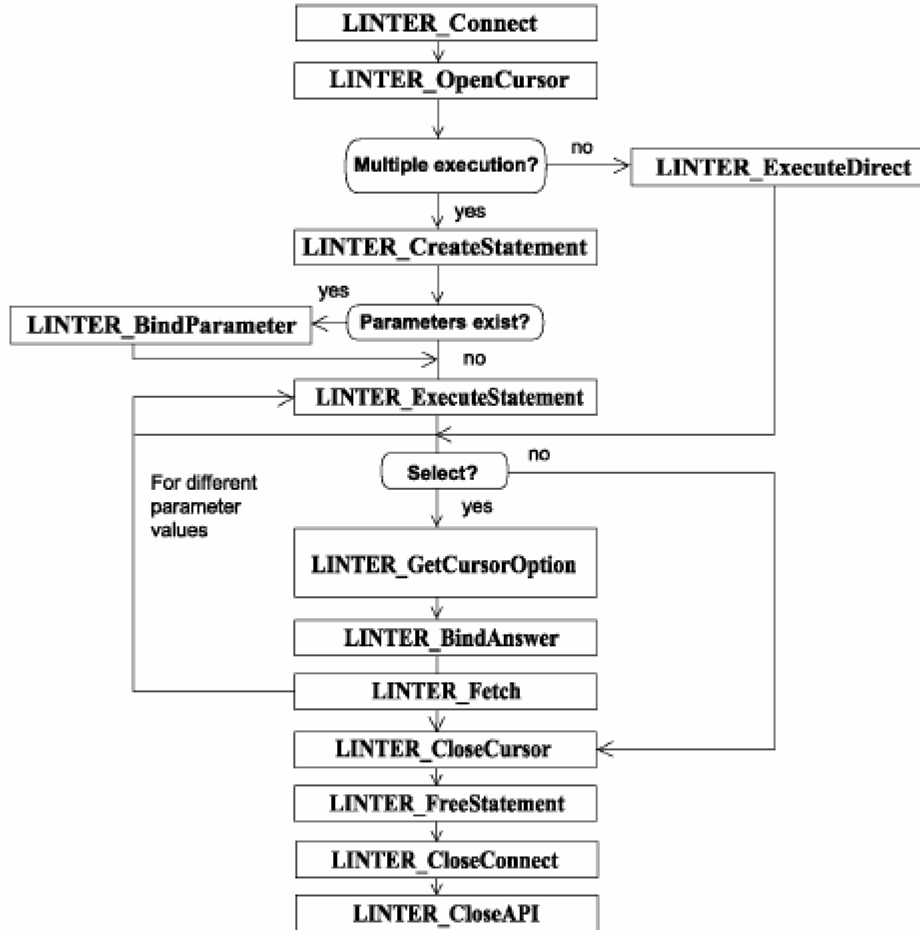
For the non-windows platforms the object library linapi.a should be used for the link process.

For Windows platform, the following files should be used as a LinAPI object library.

Standard Application Structure

The following flow chart shows the typical flow of a LinAPI application.

Let's analyze this structure.



To execute any query, you need to create a connection and open a cursor. If your query has parameters or is to be executed multiple times, you will need to create a statement. If there are parameters, they must be bound.

The next step is statement execution. If a SELECT query is being executed, answer field buffers must be bound.

The statement is executed as many times as needed.

If a query has no parameters or is to be executed only once, there is no need to create a statement. Such queries are executed directly with the LINTER_ExecuteDirect function. If a SELECT query is being executed just once, the sequence will be similar to statement processing.

Before terminating the application, it is necessary to close all cursors and connections, free all statements, and free all system resources using the LINTER_CloseAPI function.

It should be noted that the described sequence of actions is not the only one acceptable. For example, binding of answer field buffers may precede statement execution; it is also not necessary to bind the buffer to get the answer. You may use other tools, depending on the application.

Functions

Connections

Open Connection

```
long LINTER_Connect(
     char *UserName,          /* User name */
     short NameLen,           /* Len. of UserName */
     char *Password,          /* User password */
     short PassLen,           /* User password len. */
     char *ServerName,        /* Linter server name */
     long Mode,               /* Trans. Proc. Mode */
     short *ConnectID);       /* Connect ID number */
```

Input parameters

UserName, NameLen	User name and its length.
Password, PassLen	User password, which may be NULL, andits length.
ServerName	Name of Linter-server, which may, be NULL to use the default server.
Mode	Transaction processing mode.

Output parameter

ConnectID	ID number of connection.

Action

Opens a connection to the Linter server specified by `ServerName` for user `UserName` identified by the password `Password`. Multiple connections may be active at the same time.

It is possible to enter both user name and password in the `UserName` field using a slash (`Username/Password`).

For case-sensitive user name or password use the double quote character, like: "Brian"/"Davis".

`NameLen` and `PassLen` might be zero, in that case `UserName` and `Password` parameters should represent NULL-terminated strings.

`ServerName` parameter is a NULL-terminated string and in case he string is not empty the parameter value should match of the database server names located in ~linter/bin/nodetab file.

The connection will have the transaction processing mode specified by Mode.

There are three transaction processing modes in Linter: Autocommit, Optimistic and Exclusive. Identifiers for these modes are defined in header file linapi.h as mAutocommit, mOptimistic and mExclusive respectively.

Example

```
long      Error;
short     nConnID;
…
If (lError=LINTER_Connect
("system",0,"manager",0,NULL,mOptimistic,&nConnID))  processing_error(lError,
nConnID, 0, 0, "LINTER_Connect");
```

Close Connection

```
long LINTER_CloseConnect(
  short ConnectID);          /* Connection ID */
```

Input parameter

ConnectID.

Output parameters

None.

Action

Closes the ConnectID connection and all of its cursors and statements. All related to the connection id not completed transactions will be committed.

Example

```
long lError;
Short nConnID;
…
If (lError=LINTER_ConnectClose(nConnID))
Processing_error(lError, nConnID, 0, 0, "LINTER_CloseConnect");
…
```

Open connect asynchronously

```
long LINTER_AsyncConnect
     (char* Username,            /* DB User name string */
     short UID_Length,           /* User name string length */
     char* Password,             /* Password length */
     short Password_L ength,     /* Password string length */
     char*  DBServerNa me,       /* Linter DB server name */
     long Transaction Mode,      /* Connection transaction mode */
     short* ConnectionID         /* returned connection id */
     void* UserCallBackFunction  /* User callback function pointer */
     void* Parameters);          /* User callback function parameter */
```

Get Connection Attributes

```
long LINTER_GetConnectOption(
  short ConnectID,  /* ID number */
  short OptionType, /* Attrib. type */
  void *Buffer,     /* Attr. Buff. */
  long *BufLen);    /* Buff. Len. */
```

Input parameters

ConnectID	ID number of connection.
OptionType	Type of connection attribute.
BufLen	Length of attribute value buffer.

Output parameters

Buffer	Attribute value buffer.
BufLen	Actual length of returned attribute value.

Action

Writes attribute value into buffer. Almost all connection attributes are of data type long except cDBDesc, which is the same as t_DBDesc.

If BufLen is NULL, LinAPI doesn't check that bufer has enough space for the attribute value.

Example

```
long  lError;
Short nConnID;
t_DBDesc     dDbDesc;
…
If (lError=
LINTER_GetConnectOption(nConnID, cDBDesc, (void *)&dbDesc, NULL))
processing_error(lError, nConnID, 0, 0,"LINTER_GetConnectOption");
```

Set Connection Attributes

```
long LINTER_SetConnectOption(
  short ConnectID,     /* ID no. */
  short OptionType,
  void *OptionValue
  long *ValueLen);     /* Opt. length*/
```

Input parameters

ConnectID	ID number of connection.
OptionType	Type of connection attribute, currently limited to cPriority type.
OptionValue	Value of attribute, currently limited to priority.
ValueLen	Length of attribute value in bytes.

Output parameters

None.

Action

Sets connection attributes.

Example

```
long      lError;
short     nConnID;
short     nPriority = 10;
…
If (lError=
LINTER_SetConnectOption(nConnID, cPriority,&nPriority, NULL))
processing_error(lError, nConnID, 0, 0,"LINTER_SetConnectOption");
```

Cursors

Open Cursor

```
long LINTER_OpenCursor(
  short ConnectID,   /* Connect ID no. */
  short *CursorID,   /* Cursor ID no.*/
  char *CursorName,  /* Cursor name*/
  short NameLen,     /* Cursor name len.*/
  long Mode);        /* Tran.Proc. mode */
```

Input parameters

ConnectID	ID number of connection.
CursorName	Optional cursor name complying with Linter SQL name syntax. Used for sampling.
NameLen	Length of cursor name. If zero, it will be a NULL terminated character string.
Mode	Cursor's transaction processing mode. This may be different than the connection's TP mode.

Output parameter

CursorID	ID number of the opened cursor.

Action

Opens a cursor on the connection `Connect ID` with the specified transaction processing mode, assigns `CursorName`, and returns the cursor ID number. There may be multiple cursors for each connection.

If the address of `CursorName` is NULL, the cursor is considered unnamed. Wih an unnamed cursor, it is not possible to execute the SQL query UPDATE... CURRENT OF `CursorName`.

When a cursor is opened, all its attributes except prioriy are set to NULL. A cursor inherits its prioriy value from the parent connection.

Example

```
long     lError;
short     nConnID;
short     nCursID;
…
If (lError= LINTER_OpenCursor(nConnID,&nCursID, NULL, 0, mOptimistic))
processing_error(lError, nConnID, 0, 0, "LINTER_OpenCursor");
```

Close Cursor

```
long LINTER_CloseCursor(short CursorID);  /* Connect ID no. */
```

Input parameter

CursorID	ID number of the opened cursor.

Output parameters

None

Action

Closes CursorID cursor.

If some parameters or answer fields are bound to this cursor, all such bindings are terminated the moment the cursor is closed.

Example

```
long     lError;
short     nCursID;
…
If (lError= LINTER_CloseCursor(nCursID))
processing_error(Error, 0, nCursID, 0, "LINTER_CloseCursor");
```

Get Cursor Attributes

```
long LINTER_GetCursorOption(
  Short CursorID,     /* Curs. ID no. */
  Short OptionType,   /* Option type*/
  Short ColumnNumber, /* Column no.*/
  void *Buffer,       /* Buffer name */
  long *BufLen);      /* Buff. Length */
```

Input parameters

CursorID	ID number of cursor.
OptionType	Attribute type.
ColumnNumber	Number of returned column, for cAnswerDesc.
BufLen	Length of attribute value buffer.

Output parameters

Buffer	Attribute value buffer.
BufLen	Actual length of returned attribute value.

Action

Writes the value of the specified attribute to the buffer. Cursor attributes are described in section “Cursor Attributes”. Almost all cursor attributes are of type long except cAnswerDesc, which is the same as t_ParamDesc.

If BufLen is NULL, LinAPI doesn't check that bufer is large enough to accept the attribute value.

Example

```
long  lError;
short nCursID;
t_ParamDesc     pdAnsDesc;
…
If (lError=
LINTER_GetCursorOption(nCursID,cAnswerDesc,1,void*)&pdAnsDesc,NU LL))
processing_error(lError, 0, nCursID, 0, "LINTER_GetCursorOption");
```

Set Cursor Attributes

```
long LINTER_SetCursorOption(
  short CursorID,       /* Curs. ID no. */
  short OptionType,     /* Option type*/
```

```
  void *OptionValue,    /* Opt. value*/
  long *ValueLen);      /* Val. Length */
```

Input parameters

CursorID	ID number of cursor.
OptionType	Option type, only cPriority is currently supported.
OptionValue	Attribute value; currently, only priority.
ValueLen	Length of attribute value in bytes.

Output parameters

None.

Action

Sets cursor attributes.

Example

```
long      lError;
short     nCursID;
short     nPriority = 10;
…
If (lError=
LINTER_SetCursorOption(nCursID,cPriority,(void*)&nPriority,NULL))
processing_error(lError, 0, nCursID, 0, "LINTER_SetCursorOption");
```

Statements

Create Statement

```
long LINTER_CreateStatement(
  short ConnectID,       /* Conn. ID no. */
  char *Query,           /* Query text */
  long Length,           /* Query length */
  short *StmID);         /* ID number of statement*/
```

Input parameters

ConnectID	ID number of connection.
Query	Query text.
Length	Length of query text.

Output parameters

StmtID	ID number of statement.

Action

Complies a Query and creates a Statement ID.

Example

```
long     lError;
short     nConnID;
short     nStmtID
char      cQuery[ ] =
”select make,model from auto where make = ‘FORD’”;
…
If (lError= LINTER_CreateStatement(nConnID,cQuery,0, &nStmtID))
```

```
processing_error(lError, nConnID, 0, 0, "LINTER_CreateStatement");
```

Free Statement

```
Long LINTER_FreeStatement(
  short StmtID      /* Stmt. ID no. */
```

Input parameter

StmtID	ID number of statement.

Output parameters

None.

Action

Deletes the statement from the list of statements for the connection used when creating this statement.

Example

```
long lError;
short nStmtID;
if (lError =LINTER_FreeStatement(nStmtID))
  Processing_error(lError, 0, 0, nStmtID, "LINTER_CreateStatement");
```

Get Statement Attribute

```
long LINTER_GetStatementOption(
  short StatementID,    /* Stmt. ID no. */
  short OptionType,     /* Option type*/
  short ParamNumber,    /* Param. no.*/
  void *Buffer,         /* Buffer name */
  long *BufLen);        /* Buff. Length */
```

Input parameters

StatementID	ID number of statement.
OptionType	Attribute type.
ParamNumber	Number of parameters for sParamDesc.
BufLen	Length of attribute value buffer.

Output parameters

Buffer	Attribute value buffer.
BufLen	Actual length of returned attribute value.

Action

Writes the value of the specified attribute to buffer. Almost all of statement attributes are of type long except sParamDesc and sAnswerDesc, which have the same type as t_ParamDesc.

Example

```
long lErr;
short StmtID;
long lParCnt;
if(lErr=LINTER_GetStatementOption(StmtID, sParamCnt, 0,
   void *)&lParCnt,NULL))
  processing_error(lErr, 0, 0, StmtID, "LINTER_GetStatementOption");
```

Execute Statement

```
long LINTER_ExecuteStatement(
  short CursorID,      /* Cursor ID no. */
  short StatementID,   /* Stmt.ID no.*/
  long *ExecCount,     /* no. iterations*/
  void *AsyncFunc,     /* Asn. function */
  void *UserArg);      /*User Argument*/
```

Input parameters

CursorID	ID number of cursor.
StatementID	ID number of statement.
ExecCount	Number of times to execute the statement.
AsyncFunc	Answer processing function.
UserArg	Length of attribute value buffer.

Output parameters

ExecCount	Number of times statement was actually executed.

Action

Executes statement on the specified cursor.

Example

```
long lError;
short nCursID;
short nStmtID;
…
if (lError =LINTER_ExecuteStatement(nCursID, nStmtID, NULL, NULL, NULL))
  processing_error(lError, 0, nCursID, 0, "LINTER_ExecuteStatement");
```

Processing Queries and Parameters

Bind Query Parameter

```
long LINTER_BindParameter(
  short CursorID,           /* Cursor ID no. */
  short StatementID,        /* Stmt.ID no.*/
  short ParameterNumber,    /* Parameter no. */
  char *ParameterName,      /* Parameter Name*/
  char *NullIndicator,      /* Attribute is NULL */
  void *ParameterAddress,   /* ParameterAddress */
  long ItemCounter,         /* no. items in array*/
  long ItemLength,          /* Item length */
  short ParameterType,      /* Parameter type*/
  long *Reserved,           /* Not used */
  long *ParameterLength);   /* Length of parameter values */
```

Input parameters

CursorID	ID number of cursor.
StatementID	ID number of statement.

ParameterNumber	Parameter number. This is used if ParNam, is NULL. Parameter numbering begins with 1.
ParameterName	Address of NULL terminated string containing parameter name.
NullIndicator	NULL value flag.
ParameterAddress	Address of parameter value buffer.
ItemCounter	Number of parameters in array.
ItemLength	Size of one element of parameter array.
ParameterType	Parameter type.
ParameterLength	Address of parameter value length.

Output parameters

None.

Action

Binds parameter buffer to the statement. When the statement is executed, values from the buffer will be inserted into corresponding parameter places. You can create queries with parameters using LINTER_CreateStatement. Parameters may be named or unnamed.

Example using named parameter:

```
SELECT * FROM auto WHERE make = :MAKE;
```

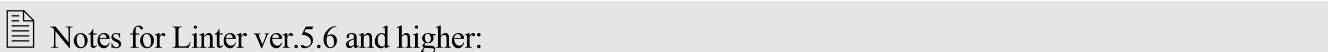
Examples using unnamed parameter:

```
1 SELECT * FROM auto where make = ?;
2 INSERT INTO auto(PersonID ,Make) VALUES (? ,?);
```

Several values may be bound to one parameter by binding the entire, ItemCnt, array of elements, sized as ItemLen.

Parameter length, ParLen, also should be presented as an array of values of type long. When LINTER_ExecuteStatement is called, the statement will be executed for all parameter values.

If a parameter is named, it may be bound by either name or number. A query may contain both named and unnamed parameters.


Notes for Linter ver.5.6 and higher:

1) There are three parameter classes:

- input: pInput;
- output: pOutput;
- input & output: pInputOutput.

2) The bit masks pInput, pOutput and pInputOutput are defined in the linapi.h file. ParType can be obtained by performing an OR operation with the corresponding bit mask (pInput, pOutput or pInputOutput) and parameter type.

3) If parameters are of type string, t_String, they will be considered as NULL-terminated strings. Earlier versions treated them as NULL-terminated string only if ParLen was -1.

Example

```
long lErr;
short nCnnID;
short nCrsID;
short nStmtID;
char cPar[] ='FORD';
long lParL;
char cQuery[]="select make from auto where make = :MAKE";
…
if(lErr=LINTER_CreateStatement(nCnnID, cQuery, 0, &nStmtID))
  processing_error(lErr, nCnnID, 0, 0, "LINTER_CreateStatement");
lParL =sizeof(cPar);
if(lErr=LINTER_BindParameter(nCrsID,nStmtID,0,"MAKE",
                               NULL,(void*)cPar,1,0,tChar, NULL,&lParL))
  processing_error(lErr, 0, nCrsID, 0, "LINTER_BindParameter");
```

Bind Answer Buffer

```
long LINTER_BindAnswer(
  short CursorID,        /* Cursor ID no. */
  short StatementID,     /* Stmt.ID no.*/
  short ColumnNumber,    /* Column no. of ans. */
  void *AnswerBuffer,    /* Answer buffer */
  char *NullIndicator,   /* NULL indic. buff. */
  long ItemLength,       /* Array item length */
  short OutType,         /* Output ans. type */
  long OutLength,        /* Output length */
  short OutPrec,         /* * Output precision /
  short OutScale,        /* Output scale */
  long *RealLength);     /* Actual length array */
```

Input parameters

CursorID	ID number of cursor.
StatementID	ID number of statement.
ColumnNumber	Number of answer columns (fields).
AnswerBuffer	Address of answer buffer.
NullIndicator	Buffer of NULL value indicators corresponding to values from AnsBuff.
ItemLength	Length of buffer element for one answer.
OutType	Data type of output field.
OutLength	Length of output field.
OutPrec	Precision for tDecimal.
OutScale	Scale for tDecimal.
RealLength	Array to receive real field lengths.

Output parameter

RealLen	Actual length of answer. Filled after LINTER_Fetch is executed.

Action

Assigns a buffer to store answers retrieved using LINTER_Fetch. This buffer may be assigned for multiple answers.

ItemLen breaks the answer buffer into equal segments. Several calls to LINTER_BindAnswer may present each piece as a complex record. For example, the following calls:

```
…
typedef char Make_Type[20];
typedef struct {
   Make_Type Make;
   long PersonID;
   } Answ_Type;
Answ_Type B[20];
…
LINTER_BindAnswer(3,7,1,B, NI, sizeof(Answ_Type),
                  tChar, sizeof(Make_Type), 0,0,NULL);
LINTER_BindAnswer(3,7,2,(char*)B +sizeof(Make_Type),
                  NI, sizeof(Answ_Type),tInt, sizeof(long),0,0,NULL);
```

or

```
LINTER_BindAnswer(3,7,1, B.Make, NI, sizeof(Answ_Type),
                        tChar,sizeof(Make_Type),0,0,NULL);
LINTER_BindAnswer(3,7,2,&B.PersonID,NI,
                        sizeof(Answ_Type),tInt ,sizeof(long), 0,0,NULL);
```

Show that the answer retrieved on cursor 3 while executing the first statement has AnswType structure. Array B may then be used in he second statement to accept records.

The answer feld may only be bound afer query compilation, if using a Statement, or afer query execution, when using LINTER_ExecuteDirect. In the second case, StatementID must be equal to zero.

Unbind Answer Buffer

```
long LINTER_UnBindAnswer(
  short CursorID,        /* Cursor ID no. */
  short StatementID,     /* Stmt.ID no.*/
  short ColumnNumber);   /* column no. Of ans. */
```

Input parameters

CursorID	ID number of cursor.
StatementID	ID number of statement.
ColumnNumber	Number of answer columns (fields).

Output parameters

None.

Action

Unbinds the buffer bound by LINTER_BindAnswer without performing any action on it. When LINTER_Fetch is next called, the result set data will not be written to this buffer. If the column number is zero, all buffers will be unbound.

Example

```
long lError;
short nCursID;
short nStmtID;
…
if (lError =LINTER_UnBindAnswer(nCursID, nStmtID, 1))
   processing_error(lError, 0, nCursID, 0, "LINTER_UnBindAnswer");
```

Transactions and Locks

Commit Changes on Cursor

```
long LINTER_CommitCursor(
  short CursorID,       /* Cursor ID no. */
  void *AsyncFunc,      /* Ans.proc.function*/
  void *UserArg);       /* User argument */
```

Input parameters

CursorID	ID number of cursor.
AsyncFunc	Address of answer processing function.
UserArg	User argument.

Output parameters

None.

Action

Confirms transaction results and writes changes to database.

Example

```
long lError;
short nCursID;
…
if (lError =LINTER_CommitCursor(nCursID, NULL, NULL))
  processing_error(lError, 0, nCursID, 0, "LINTER_CommitCursor");
```

Confirm Changes on Connection

```
long LINTER_Commit(
  short ConnectID,  /* Connection ID no. */
  void *AsyncFunc,  /* Ans.process function */
  void *UserArg);   /* User argument */
```

Input parameters

ConnectID	ID number of connection.
AsyncFunc	Address of answer processing function.
UserArg	User argument.

Output parameters

None.

Action

Confirms transaction results of all cursors for the connection ConnectID and writes changes to database.

Example

```
long lError;
short nConnID;
…
if (lError =LINTER_Commit(nConnID, NULL, NULL))
  processing_error(lError, nConnID, 0, 0, "LINTER_Commit");
```

Rollback Changes on Cursor

```
long LINTER_RollBackCursor(
  short CursorID,       /* Connect ID no. */
  void *AsyncFunc,      /* Ans.proc.func. */
  void *UserArg);       /* User argument */
```

Input parameters

ConnectID	ID number of connection.
AsyncFunc	Address of answer processing function.
UserArg	User argument.

Output parameters

None.

Action

Rolls back transaction on cursor.

Example

```
long lError;
short nCursID;
…
if (lError =LINTER_RollBackCursor(nCursID, NULL, NULL))
  processing_error(lError, 0, nCursID, 0, "LINTER_RollBackCursor");
```

Rollback Changes on Connection

```
long LINTER_RollBack(
  short ConnectID,     /* Connection ID no. */
  void *AsyncFunc,     /* Ans.process function*/
  void *UserArg);      /* User argument */
```

Input parameters

ConnectID	ID number of connection.
AsyncFunc	Address of answer processing function.
UserArg	User argument.

Output parameters

None.

Action

Rolls back transactions on all cursors on the connection.

Example

```
long lError;
short nConnID;
…
if (lError =LINTER_RollBack(nConnID, NULL, NULL))
  processing_error(lError, nConnID, 0, 0, "LINTER_RollBack");
```

Lock Row

```
long LINTER_LockRow(
      short CursorID);   /* Cursor ID no. */
```

Input parameter

CursorID ID number of cursor.

Output parameters

None.

Action

Locks the row currently selected by cursor.

Example

```
long lError;
short nCursID;
…
if (lError =LINTER_LockRow(nCursID))
  processing_error(lError, 0, nCursID, 0, "LINTER_LockRow");
```

Unlock Row

```
long LINTER_UnlockRow(
  short CursorID)   /* Cursor ID no. */
```

Input parameter

CursorID ID number of cursor.

Output parameters

None.

Action

Unlocks the row currently selected by cursor.

Example

```
long lError;
short nCursID;
…
if (lError =LINTER_UnLockRow(nCursID))
  processing_error(lError, 0, nCursID, 0, "LINTER_UnLockRow");
                  "LINTER_UnLockRow");
```

LINTER_ExecuteDirect

```
long LINTER_ExecuteDirect(
  short CursorID,   /* Cursor ID no. */
  Char *Query,      /* Ptr. To query text */
  Long QueryLength, /* Query length */
  Void *AsyncFunc,  /* Ans.proc func.addr.*/
  void *UserArg);   /* User argument */
```

Input parameters

CursorID	ID number of connection;
Query	text of query;
QueryLength	length of answer;
AsyFunc	address of answer from processing function;
UserArg	user argument. The AsyFunc can be used with a single argument.

Output parameters

None.

Action

Executes query using CursorID.

After query execution, you may call the following functions: LINTER_GetCursorOption, LINTER_BindAnswer, LINTER_Fetch, or LINTER_GetRowBuffer.

Example

```
long lError;
short nCursID;
char cQuery[] = "select make, model from auto";
…
if (lError =LINTER_ExecuteDirect(nCursID, cQuery, 0, NULL, NULL))
  processing_error(lError, 0, nCursID, 0, "LINTER_ExecuteDirect");
```

LINTER_Fetch - cursor positioning

```
long LINTER_Fetch(
  short CursorID,   /* Cursor ID number */
  short Direction,  /* Direction of movement */
  Long Position,    /* Position number */
  Long RowCounter,  /* Requirement row counter */
  Long *RowAnswer,  /* Number of rows to fetch */
  Void *AsyncFunc,  /* Ans.proc func.addr.*/
  void *UserArg);   /* User argument */
```

Input parameters

CursorID	ID number of cursor.
Direction	Fetch direction or orientation.
Position	Number of current row.
RowCounter	Number of rows to be fetched if RowCounter <= 0.
AsyncFunc	Address of answer from processing function.
UserArg	User argument. The AsyncFunc can be used with a single argument.

Output parameter

RowAnswer	actual number of rows written to answer buffer.

Action

Writes the specified number of answer rows to the answer buffer, starting with the row specified by `Direction`.

The last row fetched becomes the current row.

For Linterv44 and earler, asynchronous movement wihin a selected row set is not implemented. For those earler versions, the AsyFunc and UserArg parameters must be NULL or the eFunctionNotAvailable error wil be returned.

Options for `Direction` are:

Name	Direction	Resulting Row Number
toNext	to the next row	CurrNumber +RowAnswer
toPrevious	to the previous row	CurrNumber -1 +RowAnswer -1
toFirst	to the first row	RowAnswer
toLast	to the last row	LastNumber
toAbsNumber	to the row with specified absolute number in the answer.	Position +RowAnswer –1
toRelNumber	to the row with specified number relative to the current row.	CurrNumber +Position +RowAnswer -1
toFromEnd	from the end	LastNumber

Example

```
long lError
short nCursID;
…
if (lError =LINTER_Fetch(nCursID, toNext, 0, 20, NULL, NULL, NULL))
  processing_error(lError, 0, nCursID, 0, "LINTER_Fetch");
```

Get Answer Row to Buffer

```
long LINTER_GetRowBuffer(
  short CursorID,        /* Cursor ID no. */
  void *Buffer,          /* Answer buffer */
  long *BufLen);         /* Buffer length */
```

Input parameters

CursorID	ID number of cursor.
Buffer	Address of answer row buffer.
BufLen	Length of answer buffer.

Output parameter

`BufLen`	Actual length of rows written to answer buffer.

Action

Writes the answer row to buffer without any conversion. This function is most useful when the structure of the answer is known.

Example

```
long lError;
short nCursID;
long lAnsLen;
void *vBuffer;
…
if(lError =LINTER_GetCursorOption(nCursID,cAnswerSize,
                              0,(void*) &lAnsLen, NULL))
  processing_error(lError, 0, nCursID, 0, "LINTER_GetCursorOption");
vBuffer = calloc(lAnsLen, sizeof(char));
…
if(lError =LINTER_GetRowBuffer(nCursID, (void *) vBuffer, &lAnsLen))
  processing_error(lError, 0, nCursID, 0, "LINTER_GetRowBuffer");
```

Get Answer Column to Buffer

```
long LINTER_GetData(
  short CursorID,      /* Cursor ID number */
  short ColumnNumber,  /* Column no. in answer */
  void *OutBuffer,     /* Answer buffer */
  long OutBufLen,      /* Len of OutBuf in bytes */
  char OutType,        /* Data type of ans. field */
  char OutPrec,        /* Precision for tDecimal */
  char OutScale,       /* Scale for tDecimal */
  long *RealLength);   /* Actual length of field */
```

Input parameters

CursorID	Cursor ID number.
ColumnNumber	Number of column in answer buffer.
OutBuffer	Address of answer buffer.
OutBufLen	Length of OutBuf.
OutType	Data type of answer field.
OutPrec	Precision, only for tDecimal.
OutScale	Scale, only for tDecimal.

Output parameters

OutBuffer	Output buffer.
RealLength	Actual length of answer.

Action

Writes the value of answer column to OutBuf and the actual length of the answer to RealLen.

Example

```
long lError;
short nCursID;
t_ParamDesc pdAnsDesc;
void *vBuff;
…
if(lErr=LINTER_GetCursorOption(nCrsID,cAnswerDesc,1,
                              (void*)&pdAnsDesc,NULL))
  processing_error(lErr, 0, nCrsID, 0, "LINTER_GetCursorOption");
vBuff = calloc(pdAnsDesc.Length, sizeof(char));
…
if(lErr=LINTER_GetData(nCrsID,1,vBuff,pdAnsDesc.Length,
                       pdAnsDesc.Type,0,0,NULL))
  processing_error(lErr, 0, nCrsID, 0, "LINTER_GetData");
```

Working with BLOBs

The BLOB data functions are operated on last processed row of the cursor. The last processed row might be as a result of select/etch request, insert request, update request.

Following functions applied to the first BLOB type column in a cursor:

LINTER_AppendBlob	Add the portion of data to the end of the BLOB ype column value.

LINTER_GetBlob	Receive the portion of BLOB type column data to the application bufer.
LINTER_ClearBlob	Delete the BLOB type column data.
LINTER_GetBlobType	Retrieve user-defined BLOB type value.
LINTER_SetBlobType	Set user-defned BLOB type value.
LINTER_GetBlobLength	Retrieve the BLOB data length.

In case of select query, the BLOB column should be listed in a result set. In case of insert or update queries as a last processed queries on a cursor, the listed BLOB functions will be applied to the first BLOB column in table schema. The listed BLOB functions are provided as a compatibly functions with the previous versions of RELEX DB Linter Series.

In case if the table schema or the processed row contain more than one BLOB type column, the following functions should be used in order to operate with the BLOB type column data:

LINTER_PurgeBlob	Delete the BLOB data for the specied column.
LINTER_AddBlob	Add the data portion to BLOB column data specied by number.
LINTER_FetchBlob	Fetch BLOB data of specied column into the application bufer.

The operations for the listed above BLOB functions will be applied to the BLOB column using the BLOB column number in a list of the BLOB columns of the last processed query. For example, the last processed row is a result of fetch for "select make, description, model, picture from auto;". The column data type of the query result set respectively is:

- Make - character(20);
- Description – BLOB;
- Model - character(20);
- Picture – BLOB.

There are two BLOB columns and to address the "Description" BLOB data application has to use 1 as a BLOB column number and 2 for "Picture" column.

Append Data to BLOB

```
long LINTER_AppendBlob(
  short CursorID,       /* Cursor ID number */
  short Blob_Type,      /* BLOB value type */
  void *Buffer,         /* New data buffer */
  long BufLen,          /* Len of Buff in bytes */
  void *AsyncFunc,      /* Ans. process func. */
  void *UserArg);       /* user argument */
```

Input parameters

CursorID	Cursor ID number.
Blob_Type	Type of BLOB data.
Buffer	Address of buffer containing new BLOB data.
BufLen	Length of Buffer and new BLOB data.

AsyncFunc	Address of answer from processing function.
UserArg	User argument.

Output parameters

None.

Action

Adds a segment of BLOB data to the BLOB data of the current row.

Example

```
long lError;
short nCursID;
long lBufLen;
void *vBuffer;
…
if(lError =LINTER_AppendBlob(nCursID, 0, vBuffer, lBufLen, NULL, NULL))
  processing_error(lError, 0, nCursID, 0, "LINTER_AppendBlob");
```

Select portion of BLOB data

```
long LINTER_GetBlob(
  short CursorID,   /* Cursor ID number */
  long OffSet,      /* Position in Blob data */
  long *Size,       /* User buffer size */
  void *Buffer,     /* User buffer pointer */
  void *AsyncFunc,  /* User-defined callback function.*/
  void *UserArg);    /* User callback function parameters */
```

Input parameters

CursorID	Cursor ID number.
OffSet	Offset in blob data. The offset begins with value 1.
Size	User buffer size.
Buffer	User buffer pointer.
AsyncFunc	Pointer to user callback function. Used for asynchronous execution.
UserArg	User parameters for user callback function.

Output parameters

Buffer	Pointer to user buffer for the BLOB data.
Size	BLOB data user buffer length.

Action

Retrieve portion of the BLOB data with specified offset into the application buffer.

Example

```
long     Error;
short    nCursID;
long     lBufLen;
void*    vBuff;
…
If (lError= LINTER_GetBlob(nCursID, 1, vBuff, lBufLen, NULL, NNULL))
processing_error(lError, 0, nCursID, 0, "LINTER_GetBlob");
```

Clear BLOB data

```
long LINTER_ClearBlob(
  short CursorID,     /* Cursor ID number */
  void *AsyncFunc,    /* Ans. processing func.*/
  void *UserArg);     /* User argument */
```

Input parameters

CursorID	Cursor ID number.
AsyncFunc	Address of answer from processing function.
UserArg	User argument.

Output parameters

None.

Action

Clears BLOB data from current row.

Example

```
long lError;
short nCursID;
…
if (lError =LINTER_ClearBlob(nCursID, NULL, NULL))
  processing_error(lError, 0, nCursID, 0, "LINTER_ClearBlob");
```

Get or Set BLOB Type

```
long LINTER_GetBlobType(
  short CursorID,     /* Cursor ID number */
  short *Blob_Type);  /* Type of BLOB data */

long LINTER_SetBlobType(
  short CursorID,     /* Cursor ID number */
  short Blob_Type);   /* Type of BLOB data */
```

Input parameters

CursorID	Cursor ID number.
Blob_Type	Type of BLOB value, see Linter_PutBlobType.

Output parameters

Blob_Type	Type of BLOB value.

Action

LINTER_GetBlobType returns the type of BLOB value. LINTER_SetBlobType changes the type of BLOB value.

Example

```
long lError;
short nCursID;
short nType;
…
if(lError =LINTER_GetBlobType(nCursID, &nType))
  processing_error(lError, 0, nCursID, 0, "LINTER_GetBlobType");
```

Get Size of BLOB

```
long LINTER_GetBlobLength(
  short CursorID,    /* Cursor ID number */
  long *Blob_Size);  /* BLOB data length*/
```

Input parameter

CursorID	Cursor ID number.

Output parameter

Blob_Size	Length of BLOB value in bytes.

Action

Writes the length of the BLOB data to `Blob_Size`.

Example

```
...
long lError;
short nCursID;
long lBlobLength;
…
if(lError =LINTER_GetBlobLength(nCursID, &lBlobLength))
  processing_error(lError, 0, nCursID, 0, "LINTER_GetBlobLength");
```

Purge blob data for specified by number blob column

```
L_lONG LINTER_PurgeBlob(
  short CursorID,     /* Cursor ID number */
  short ColumnNumber, /* Blob column number for last processed row */
  void *AsyncFunc,    /* User-defined callback function*/
  void *UserArg);     /* User Parameters */
```

Input parameters

CursorID	Cursor ID number.
ColumnNumber	BLOB column number for last processed row.
AsyncFunc	Pointer to user callback function. Used for asynchronous execution.
UserArg	User parameters for user callback function.

Output parameters

None.

Action

A function clears BLOB data for the specified column of the last processed row.

Example

```
long lError;
short nCursID;
void* vBuff;
…
If (lError= LINTER_PurgeBlob(nCursID, 2 NULL, NNULL))
processing_error(lError, 0, nCursID, 0, "LINTER_PurgeBlob");
```

Add BLOB data to specified by number BLOB column

```
L_LONG LINTER_AddBlob(
  short CursorID,     /* Cursor ID number */
  short ColumnNumber, /* Blob column number for last processed row */
```

```
 short Blob_Type,    /* User-defined blob data type*/
 void *Buffer,       /* Blob data buffer for the blob column data to
                     be added */
 Long BufLen,        /* Blob data length */
 Void *AsyncFunc,    /* User-defined callback function*/
 Void *UserArg);     /* User parameters */
```

Input parameters

CursorID	Cursor ID number.
ColumnNumber	BLOB column number for last processed row.
Blob_Type	User-defined BLOB data type (0..255 values).
Buffer	User BLOB data buffer pointer.
BufLen	To be processed BLOB data portion size.
AsyncFunc	Pointer to user callback function. Used for asynchronous execution.
UserArg	User parameters for user callback function.

Output parameters

None.

Action

A function adds BLOB data for the specified column of the last processed row. The data will be added to the end of the existing data in the BLOB column.

Example

```
#define GifFile 3
long lError;
Short nCursID;
void* vBuff;
Long BufferLength = 45893;
…
If (lError= LINTER_AddBlob(nCursID, 2 , GifFile, vBuff, BuffLength,
NULL,NULL))
processing_error(lError, 0, nCursID, 0, "LINTER_PurgeBlob");
```

Retrieve portion of specified BLOB column

```
L_LONG LINTER_FetchBlob(
 short CursorID,     /* Cursor ID number */
 short ColumnNumber, /* Blob column number for last processed row */
 long OffSet,        /* Blob data start position */
 long *Size,         /* Blob portion size pointer *
 void *Buffer,       /* Blob data buffer for the blob column data to
                     be retrieved */
 void *AsyncFunc,    /* User-defined callback function*/
 void *UserArg);     /* User parameters */
```

Input parameters

CursorID	Cursor ID number.
ColumnNumber	BLOB column number for last processed row.
OffSet	BLOB data start position.

Size	To be processed BLOB data portion size.
Buffer	User BLOB data buffer pointer.
AsyncFunc	Pointer to user callback function. Used for asynchronous execution.
UserArg	User parameters for user callback function.

Output parameters

None.

Action

A function retrieves BLOB data for the specified column of the last processed row into user buffer. The initial position of the data in BLOB column is number l (one).

Example

```
long lError;
short nCursID;
long Position = 102400;
void* vBuff;
long BufferLength = 45893;
…
If (lError= LINTER_FetchBlob(nCursID, 2 , Position, &BufferLength, vBuff,
NULL, NULL))
processing_error(lError, 0, nCursID, 0, "LINTER_FetchBlob");
```

LinAPI Messages

Get Message Text

```
long LINTER_ErrorMessage(
  long ApiError,      /* LinAPI error */
  long LinError,      /* Linter error */
  char *Message,      /* Message text */
  Short *MessLen);    /* Length of message */
```

Input parameters

ApiError	LinAPI message code.
LinError	Linter completion code.
MessLen	Length of message buffer.

Output parameters

Message	Message buffer.
MessLen	Message length.

Action

Writes messages text corresponding to specified completion codes to the message buffer.

Obsolete

This function exists only for compatibly with previous LinAPI versions. Replaced by LINTER_Error.

Get Linter Completion Code

```
long LINTER_Error(
  short ConnectID,      /* Connection ID no. */
  short CursorID,       /* CursorID no. */
  short StatementID,    /* Statement ID no. */
  long *ApiCode,        /* LinAPI error code */
  long *LinCode,        /* Linter error code */
  long *SysCode,        /* OS error code */
  char *Message,        /* Message buffer */
  short *MessLen);      /* Length of message */
```

Input parameters

ConnectID	Connection ID number.
CursorID	Cursor ID number.
StatementID	Statement ID number.
MessLen	Length of message buffer.

Output parameters

ApiCode	LinAPI message code.
LinCode	Linter completion code.
SysCode	Operating system call completion code.
Message	Address of message buffer.
MessLen	Length of message buffer.

Action

Writes completion codes of LinAPI, Linter, and the operating system and corresponding messages to output parameters if they are not NULL.

The `OCode` parameter has different meanings in compilation and execution contexts. For query compilation errors (Linter error codes from 2444 to 2999), `OCode` contains the line number and position of the first error in the query text. For query execution errors (Linter error codes 3454. 3452,13454, and 3455, `OCode` contains the number of the parameter that caused the error.

If it is necessary to get an error on a single object (connection, cursor, or statement), you must specify its ID and set other identifiers to zero.

The `LCode` and `OCode` parameters contain usefull information only if LinAPI completion code is equal to eLinterError.

It is advisable to have an error processing function in every user application.

Example

```
void processing_error(
  long ret_cod,         /* LinAPI return code */
  short con_id,         /* Connection ID */
  short cur_id,         /* Cursor ID */
  short stmt_id,        /* Statement ID */
  char *message)        /* User message */
{
  long lRet;
  long apierr =0        /* LinAPI code */
  error =0              /* Linter code */
  syserr =0             /* OS code */
```

```
    /* error is retrieved only for one object */
    if(ret_cod ==LINAPI_ERROR )
    /* получение кодов ошибок по требуемому объекту */
      if(lRet =LINTER_Error(con_id,cur_id,stmt_id,
                            &apierr,&error,&syserr,NULL,
                            NULL))
        printf("\n Diagnostic error: %ld",lRet);
      else
      {
        printf("\n ApiErr=%ld,LinErr=%ld,SysErr=%ld\n%s",
                                apierr,error,syserr,
                                message);
        if ((apierr ==eLINTERError) && error > 2000 &&
            error < 3000)
        {
        /* retrieving error codes for the specified object */
          printf("\n Syntax error: line %d, position %d\n",
                 (short)syserr, *(short*)((char*)&
                 syserr +2));

      }
    }
    else
      printf("\n Return code = %ld",ret_cod);
}
```

Check Connection Operation Completed

```
long LINTER_ConnectComplete(
  short ConnectID,  /* Connect ID no. */
  short *IsComplete /* Completion flag */
  ,
  long *ApiCode,    /* LinAPI err code */
  long *LinCode,    /* Linter err code */
  long *SysCode);   /* OS err code. */
```

Input parameter

ConnectID	Connection ID number.

Output parameters

IsComplete	Flag indicating operation complete.
ApiCode	LinAPI completion code.
LinCode	Linter completion code.
SysCode	Operating system completion code.

Action

Sets the `IsComp` flag if operation on connection is completed; otherwise, clears it. If `IsComp` is set, the variables `ACode`, `LCode` and `OCode` contain corresponding codes.

Check Cursor Operation Completed

```
long LINTER_CursorComplete(
  short CursorID,     /* Cursor ID no. */
  short *IsComplete,  /* Completion flag */
  long *ApiCode,      /* LinAPI err code */
  long *LinCode,      /* Linter err code */
  long *SysCode);     /* OS err code. */
```

Input parameter

CursorID	Cursor ID number.

Output parameters

IsComplete	Flag indicating operation complete.
ApiCode	LinAPI completion code.
LinCode	Linter completion code.
SysCode	Operating system completion code.

Action

Sets the `IsComp` flag if operation on connection is completed; otherwise, clears it. If `IsComp` is set, the variables `ACode`, `LCode` and `OCode` contain corresponding codes.

Example

```
short nCrsID;
short nIsComplete;
long lErr, lApiErr, lLinErr, lSysErr;
…
if(lErr =LINTER_CursorComplete(nCrsID,
                                  &nIsComplete,&lApiErr, &lLinErr,&lSysErr))
   processing_error(lError, 0, nCursID, 0, "LINTER_GetBlobLength");
Else
   if(!nIsComplete) printf("\n Not complete");
   else
      if(!lApiErr ) printf("\n Complete");
      else
         printf("\n Error: API %ld, LINTER %ld, System %ld",
                lApiErr,lLinErr,lSysErr);
```

Get DB Object Description

```
long LINTER_GetObjDesc(
   short ConnectID,  /* Connect ID no. */
   Long ObjType,     /* Object type */
   Char *ObjName,    /* Object name */
   short NameLen,    /* Name length */
   Void *Buffer,     /* Description buffer */
   Long *BufLen);    /* Length of buffer */
```

Input parameters

ConnectID	Connection ID number.
ObjType	Object type, only oView is currently possible.
ObjName	Name of object, conforming to Linter syntax.
NameLen	Length of object name.
BufLen	Length of object description buffer.

Output parameters

Buffer	Address of buffer holding object description.
BufLen	Actual length of returned object description.

Action

Writes the description of the specified object to buffer.

End Session

```
long LINTER_CloseAPI (void);
```

No parameters.

Action

Closes all active connections, cursors, and statements and frees all allocated resources.

Close database channel

```
long LINTER_KillChannel (
  short ConnectID,  /* Connect ID no. */
  Short ChannelID); /* Object type */
```

Input parameters

ConnectID	Connection ID number.
ChannelID	Database channel id to be closed.

Output parameters

None

Action

A function forces to close the database channel. Linter database keeps all the connections and the cursors in the Linter internal structure anemd as channel. Channel as a synonym connection, but used for cursor as well.

Retrieve database server information

```
L_long LINTER_ServerInfo(
  char* ServerName,        /* DB Server name. */
  short ServerNameLegth,   /* DB server name length */
  short InfoType,          /* DB server info type */
  void* Buffer,            /* pointer to information buffer */
  Short BufferLength,      /* information buffer length*/
  Short *OutLengthPtr,     /* output length for ngth information
                           buffer */
  long* ApiCode,           /* LinAPI error */
  long* LinCode,           /* Linter error return */
  long* SysCode);          /* OS return code */
```

Input parameters

ServerName	Database server name listed by nodetab file.
ServerNameLength	Database server name length.
InfoType	Database server information type.
BufferLength	Information buffer size.

Output parameters

Buffer	Pointer to the information buffer.
OutLengthPtr	Returned information size.
ApiCode	Pointer to the Linter API completion code.

LinCode	Pointer to the Linter DBMS completion code.
SysCode	Pointer to the OS completion code.

Action

A function retrieves the information regarding the database server based on database server name. The database server name should be listed in ~linter/bin/nodetab file.

Cancel request processing

```
L_LONG LINTER_Cancel(
  short ConnectionID,  /* Connect ID no. */
  short CursorID,      /* Cursor ID */
  lond Option,         /* Cancel option */
  long Reserved);
```

Input parameters

ConnectionID	Connection ID number.
CursorID	Cursor ID of the processing query to be canceled.
Option	Following options are available for Cancel function: • C_CANCEL; • C_CANCEL_AST; • C_CANCEL_AST_CLOSE_CH.
Reserved	Reserved for future use.

Output parameters

None.

Action

A function cancels the user request processing by the specified cursor id. Options to be used in conjunction with Cancel function are following:

- C_CANCEL – cancel query with no additional actions. Then C_CANCEL bit is set, and then other bits are ignored;
- C_CANCEL_AST – cancel query with the immediate call of user defined callback function. The Cursor status allows to continue executing queries after the Linter response will be received;
- C_CANCEL_AST_CLOSE_CH – that option cancels the user defined function future calls and closes cursor.

Programming Hints

Asynchronous Programming

Applications created with LinAPI may use asynchronous query execution and answer processing. An asynchronous function call lets program execution continue pending call completion. You can define your own answer processing functions in which program execution is interrupted after call completion, and the answer processing function is called.

Working asynchronously is particularly useful when the application performs not only database processing, but also when:

1) other actions such as graphic output or reading information from a mass storage device are needed;
2) the application concurrently executes several independent queries;
3) the application will be submitting queries that require lengthy execution times.

The Linter distribution contains a sample asynchronous data loader.

Asynchronous programming requires careful application design because asynchronous calls are difficult to debug. All LinAPI functions are re-entrant; therefore, asynchronous execution of these functions cannot cause data damage. The application programmer must insure that his answer processing functions are re-entrant, since there is no guarantee hat when such a function is processed asynchronously, another or the same answer processing function will be called.

In case of incorrect application behavior during asynchronous processing, the address of the answer processing function can be lost and he answer processing function can not then be called. To avoid such a situation, applications can call the LINTER_ConnectComplete or LINTER_CursorComplete functions when waiting too long for an answer from an asynchronous call. If the IsComp flag is set after a completion check, then an answer has been received from Linter, but the answer processing function has not been called. This means that the application is in an invalid state and must be terminated.

A common asynchronous programming technique is to create an implicit loop of asynchronous calls. If the answer processing function contains an asynchronous LinAPI function call that species the address of the same answer processing function as an argument, an implicit loop of asynchronous calls will created.

This technique can be viewed as though the first asynchronous LinAPI creates a thread that will interrupt main program execution only for answer processing. The asynchronous LinAPI function call must be he last operator in the answer processing function. Otherwise, the answer processing function will be incomplete when the next answer from Linter is received, resulting in an overflow of the asynchronous call queue.

Answer processing functions cannot contain synchronous LinAPI functions. If an application does contains synchronous LinAPI unction calls in the answer processing function, the results may be chaotic.

Working with Stored Procedures

LinAPI contains a full set of functions needed for working with Linter stored procedures. LinAPI allows creating procedures, processing procedure compilation errors, executing procedures, and receiving answers from procedures, including cursors.

To execute/create stored procedure the regular LinAPI calls like LINTER_ExecuteDirect calls should be used.

Processing Compilation Errors

If stored procedure creation has been completed with a compilation error, the Linter completion code 7200, procedure compilation error, will be received.

Stored procedure compilation may cause several compilation errors. To obtain all compilation errors, you must repeatedly call the LINTER_Error function until receiving LinAPI completion code eNoMoreErrors.

To get the numbers of the compilation errors, use LINTER_GetCursorOption for the cursor attribute cProcErrNum.

To get the error with the specified number, you should first call LINTER_SetCursorOption for the cursor attribute cProcErrNum and then call LINTER_Error. The next LINTER_Error call will then return the next compilation error. You can receive compilation errors repeatedly and in any order.

Processing Returned Values and Output Parameters

You can get values returned by procedures and output parameter values in three different ways:

1) with the LINTER_GetData function;
2) with the LINTER_BindParameter function, if using precompiled queries with parameters;
3) using stored procedures to process answers.

LINTER_GetData can be used when either the procedure call query was precompiled or when it was executed directly. After procedure execution, you can obtain the number of output parameters with LINTER_GetCursorOption from the cursor attribute cProcArgNum. The returned value is number zero and is not counted in number of output parameters. To obtain a description of the returned value and output parameters, call LINTER_GetCursorOption for the cursor attribute cAnswerDesc.

If the procedure was called with debugging, you can get the output parameter name by calling LINTER_GetCursorOption for the cursor attribute cProcArgName if the value includes f Name flag.

If returned value flags are needed, you can get these flags by calling LINTER_GetCursorOption for the cursor attribute cProcArg Flags.

LINTER_BindParameter is used, when a statement was created for execution with parameters, to bind both input and output parameters. When binding parameters, you must specify the parameter class (pInput, pOutput, or pInputOutput) along with the parameter type. If parameter is of class pOutput or pInputOutput, bound parameter buffers will contain output parameter values after query execution.

Using stored procedures is the third method of obtaining returned values. First call LINTER_GetRowBuffer and then use low level application procedures to process the answer. See documents “The Linter Desktop Utility” & “Spdebug” for creation and debugging of stored procedure.

When Returned Value is of the Cursor Type

If the returned value has fCursor flag, it is a cursor. In this case LINTER_GetData will return the cursor ID. The cursor ID value must be of type tSmallint or tInteger. The returned ID can be used as any other LinAPI cursor ID. You can use this cursor ID in any LinAPI function call and get any of the cursor attributes except cCursorMode, cTransMode and cPriority. When the cursor is no longer needed, it should be closed with LINTER_CloseCursor.

LinAPI Completion Codes

Name	Message
eLinterError	Linter error encountered.
eNoMemory	Not enough memory for LinAPI operation.
eNullPointer	Invalid (NULL) pointer.
eIllegalSequence	Illegal sequence of actions (e.g., GetData before Execute).
eSmallBuffer	Buffer is too small.
eIllegalDirection	Illegal direction code.
eFunctionNotAvailable	Function is not supported in current version of Linter.
eIllegalParamNumber	Illegal number of parameters.
eIllegalParam	Specified parameter not found.
eIllegalBlobOperation	Illegal operation for BLOB column.
eExistsActiveChannel	Active channel exists.
eMaxIDNUmberExceeded	Too many LinAPI objects created.
eInvalidContext	Fatal internal error.
eErrorInternalDiagnostic	Internal diagnostic error.
eConnectNotFound	Specified connection not found.
eTooLongUserName	User name is too long.
eTooLongPassWord	User password is too long.
eIllegalConnectOption	Illegal connection attribute type.
eConnectIsBusy	Connection is busy.
eCursorNotFound	Specified cursor not found.
eIllegalCursorOption	Illegal cursor attribute type.
eCursorIsBusy	Cursor is busy.
eStatementNotFound	Specified statement not found.
eIllegalStatementOption	Illegal statement attribute type.
eInvalidConnectID	Invalid connection ID.
eStatementIsBusy	Statement is busy.
eQueryTooLong	Query is too long (longer than 4096 bytes).
eQueryNotSelect	Query is not a SELECT query.
eNoBlobColumn	BLOB column is missing.
eIllegalObjectType	Illegal object type.
eObjectNotView	Object is not a view.
eNoMoErrors	All errors processed (when processing procedure compilation errors).
eNoProcArgName	Unknown name of procedure argument.

Characteristics of LinAPI Objects

Connection Attributes

Attribute type	Description
cDBDesc	Description of a database.
cPriority	Connection priority.
cApiCode	LinAPI completion code.
cLinCode	Linter completion code for the last operation executed on Connection.
cSysCode	The operating system’s call completion code.
cStrNumber	Error line number for the query compilation error.
cPosNumber	Error position for the query compilation error.
cTransMode	Transaction processing mode for connection.
cNodeName	Linter server name.
cWaitComplete	Wait for completion of asynchronous operation.

Cursor Attributes

Attribute type	Description
cStrNumber	Error line number for query compilation error.
cPosNumber	Error position for query compilation error.
cCursorName	Cursor name.
cNullIndicator	NULL value flag.
cIsAutoinc	AUTOINC flag.
cProcErrNum	Number of errors encountered during procedure.
cProcArgNum	Number of procedure arguments.
cProcArgDesc	Description of procedure argument.
cProcArgName	Name of procedure argument.
cProcArgFlags	Flags of values returned by procedure.
cWaitComplete	Wait for asynchronous operation completion.

Statement Attributes

Attribute type	Description
sParamCount	Number of parameters in a statement.
sAnswerDesc	Answer description.
sColumnCount	Number of answer columns.

Attribute type	Description
sParamDesc	Parameter description.
sConnectID	ID of connection used for statement creation.
sApiCode	LinAPI completion code.
sIsAutoinc	AUTOINC flag.
sParamType	Parameter class (pInput, pOutput, pInputOutput).

LinAPI Data Types

Attribute type	Description
tChar	Character string; may contain null char.
tByte	Byte string.
tString	NULL terminated character string.
tSmallint	2-byte integer.
tInteger, tInt	4-byte integer.
tReal	4-byte real.
tDouble	8-byte (double precision) real.
Numeric, tDecimal, tDec	Real; these are type DECIMAL in Linter.
tDate, tTimeStamp	Date + time; these are DATE type in Linter.
tBlob	BLOB value.