XL Developer Guide

Preface

IMPORTANT USER INFORMATION

XL products are not designed or intended for control applications, and MUST NOT be used for control applications under any circumstances. There are fundamental differences in the design methodology of a control product such as a Programmable Logic Controller (PLC) and a non-control product such as an XL device. Outputs (e.g. relays) are provided for annunciation only, and MUST NOT be used for control purposes.

This product is designed and intended for use solely in indoor industrial applications, and must be installed by a qualified electrician.

It is the responsibility of all persons applying this product to a given installation and/or application to carefully review the installation and/or application to evaluate and ensure the suitability of this product for the intended application.

This documentation, including any examples, diagrams, and drawings, is intended to provide information for illustrative purposes only. Because of the differences and varying requirements of different installations and applications, Vorne Industries, Inc. cannot assume responsibility or liability for actual use, including use based on any examples, diagrams, and drawings.

In no event will Vorne Industries, Inc. be responsible or liable for indirect or consequential damages resulting from the use or application of this product. Please carefully review the Vorne Product Warranty Statement at www.vorne.com/warranty.htm and the Vorne Sales Terms and Conditions at www.vorne.com/terms.htm. Vorne Industries, Inc. makes no warranties express or implied except as expressly stipulated in our Product Warranty Statement.

While the information in this document has been carefully reviewed for accuracy, Vorne Industries, Inc. assumes no liability for any errors or omissions in the information.

Vorne Industries, Inc. reserves the right to make changes without further notice to any products described in this documentation.

iPhone™ is a trademark and iPod® is a registered trademark of Apple Inc. Microsoft®, Windows®, Excel®, Word®, and Internet Explorer® are registered trademarks of Microsoft Corporation. Firefox® is a registered trademark of the Mozilla Foundation. Vorne™, XL™, Productivity Appliance™ and Visual OEE™ are trademarks of Vorne Industries, Inc. All other trademarks are the property of their respective owners.

XL Data Access

OVERVIEW

All information contained in this chapter of the XL Developer Guide is subject to change upon release of new firmware versions.

The XL device stores a broad range of real-time and historical data, and it makes that data programmatically accessible through the use of a limited subset of SQL (Structured Query Language). Information about how to perform that programmatic access is split into four sections:

- XL Data Organization
- SQL Basics
- XL SQL over HTTP POST
- XL SQL Dialect

This document assumes that you have read and are familiar with portions of the *XL User Guide*, and that you know how to make an HTTP request in your programming language of choice.

XL DATA ORGANIZATION

Overview

All data on the XL device is organized into tables, and there are two kinds of table available to the end-user: **Registers** and **Streams**.

- Register tables present up-to-the-second real-time data.
- Stream tables contain historical data.

Each cell in every table contains data with one of the following types:

Data Type	Description
`int8`	8-bit signed integer (-128 to 127).
`int16`	16-bit signed integer (-32,768 to 32,767).
`int32`	32-bit signed integer (-2,147,483,648 and 2,147,483,647).
`int64`	64-bit signed integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807).
`uint8`	8-bit unsigned integer (0 to 255).
`uint16`	16-bit unsigned integer (0 to 65,535).
`uint32`	32-bit unsigned integer (0 to 4,294,967,295).
`uint64`	64-bit unsigned integer (0 to 18,446,744,073,709,551,615).
`double`	IEEE-754 double precision (64-bit) floating point number.
`date_time`	A specific point in time. In the database, this is stored in three fields. The first is the number of seconds since midnight, January 1, 1900; this value is a local time, offset from UTC. The second encodes the time offset from UTC as a number of seconds. The offset and day columns are not listed in the data schema, and are named as the `date_time` column, with `_offset` or `_day` appended. (e.g., `start_time` as specified in the schema would have a `start_time` column in the database, a `start_time_offset` column, and a `start_time_day` column.) The `offset` value is measured in seconds, and the `day` value is measured in number of days since January 1, 1900.
`time_span`	A time duration, encoded as a number of fractional seconds. These can be negative.
`enum`	Named values with integer equivalents. The numeric value is stored in the database, but the enumeration value name may be obtained by used of the SQL function

	`vorne_enum_lookup()`, described below, or by reference to the data schema.
`string`	Multilingual capable text string of arbitrary length. Length restrictions may be placed on any string in the data schema. When obtained using the XL SQL interface, strings are delimited as appropriate for the selected output format.
`blob`	Binary data of arbitrary length, encoded as a string in Base64 format.

A full XML description of the XL Data Schema may be obtained from any XL device at:

```
http://[device address]/data/schema/expanded
```

Registers

Registers store real-time data values that are generated internally (e.g. by the XL Production Monitor) or externally (e.g. set via a program or a WPI form). While some registers are updated as often as several thousand times per second, the database representation of them is updated only once per second. While some registers contain data that may be modified by the user through the WPI or XL Programming Platform, no data may be modified through the XL SQL Interface at the present time.

The following register tables are available:

Name	Data Source	Description
Intervals	Internal	Contains the metrics described as “shift-related” and “job-related” in *Production Monitor*, in two rows. The rows are differentiated by having either `job` or `shift` in the `type` column
General	Internal	Contains the metrics described as “general” in *XL User Guide > Production Monitor*, in one row.
User Numbers	External	Contains 255 integer registers in a single row. These registers are intended for entirely user-defined use through the XL Programming Platform and WPI.
User Parameters	External	Contains an assortment of user-modifiable registers that affect the functioning of the unit, but that do not rise to the level of configuration. Examples include takt time and input scale factors.
Short User Strings	External*	Contains 240 general string registers of length 16, as well as several other specific string registers with values that are generated internally. All short user string registers are in a single row.
Long User Strings	External	Contains 128 general string registers of length 256, in a single row.
Extended User Strings	External	Contains 64 general string registers of length 1024, in a single row.

Streams

Streams store all of the historical data on the XL Device. Some streams contain data that may be modified by the user through the WPI or XL Programming Platform; at the present time, no data may be modified through the XL SQL Interface.

All streams have some shared qualities:

Maximum Rows– Each has a nominal maximum number of rows available; as new records are added to the stream above this number, old records are automatically deleted. The number of rows available for the stream in the database may temporarily exceed this number by a small amount.

Sequence Number– Each stream has a `sequence_number` column that provides a unique identifier for each row in that stream. The sequence number is monotonically increasing: within a stream, a new row will always have a sequence number that is one greater than the sequence number of the row immediately before it. (Sequence numbers are large enough that even if a new one was generated every millisecond, they wouldn't run out for over 8000 years. You need not be concerned about rollover.)

The following stream tables are available:

Name	Data Source	Max Rows	Description
Interval Stream	Internal	2000	Contains historical versions of the metrics described as “shift-related” and “job-related” in *XL User Guide > Production Monitor*. A new row is created whenever a Job or Shift Reset is performed on the device, and two rows (one Job, one Shift) are created when a Master Reset is performed.
Timeline Stream	Internal*	7500	Contains both historical and current information about the Production State of the device, including a user-modifiable Reason Code. A new row is created any time the production state changes.
Diagnostic Stream	Internal	1000	Contains diagnostic events that have been reported by the XL Device firmware. This stream is usually not of interest to users.

SQL BASICS

Overview

If you’re already familiar with SQL, you can skip this section and read XL SQL Dialect below, which contains a description of differences between the SQL used with XL devices and standard SQL.

A major benefit of organizing information into tables on the XL device is that there is a standardized, well-known, and powerful query language for tabular data known as SQL (Structured Query Language). SQL was first developed by IBM to serve as an interface to relational databases, and is now a de facto standard as well as an official ANSI and ISO standard.

The XL device allows external users to query data using the SQL `SELECT` statement, which supports the following key features:

- Projection (determines which columns to return in the result-set)
- Selection (determines which rows to return in the result-set based on user-defined expressions)
- Sorting (ordering the rows of the result-set based on any combination of columns)
- Aggregation (grouping and summarizing the result-set based on user-defined criteria)

This section provides an overview of the `SELECT` statement as it relates to XL, with the goal of demonstrating the broad range of functionality that can be achieved. External access to the XL device’s onboard database is limited to the `SELECT` command; no other SQL commands are available.

The general format of the `SELECT` statement as provided by the XL device is:

```
SELECT columns FROM table [WHERE predicate-expression]
    [GROUP BY columns [HAVING predicate-expression]] [ORDER BY columns] [OFFSET number] [LIMIT number];
```

The following sections will explain the various pieces of this command and how they relate to the key features listed above.

Basic Pieces

This section covers some general SQL concepts including:

- Literals
- Expressions

Literals include numeric literals and string literals. Numeric literals have no special delimiter, but string literals are enclosed in single-quote characters. If a single-quote character occurs within a string literal, it is replaced with a sequence of two single-quote characters.

Some examples of numeric literals:

```
42
56
100.0
84397.5923
```

Some examples of string literals:

```
'job'
'Machine''s Feeder Jammed'
'Printer on Fire'
```

Expressions combine operators and values and can be used to create calculated columns, search for specific column values, filter the rows in a result-set, etc. Operators include mathematical operators such as addition, subtraction, multiplication, and division; string operators such as concatenation; and logical operations such as boolean AND and OR. Values can come from literals and/or column references.

Projection

Projection selects which columns to return in the result-set. The `SELECT` clause in conjunction with the `FROM` clause is used to specify the desired columns for the result-set. Multiple columns can be specified with a comma delimited list, and functions such as `round(column)` or mathematical expressions such as `column + other_column` are treated as columns in this clause.

```
SELECT column-name FROM table
SELECT column-name, other-column-name, ... FROM table
SELECT function(column) FROM table
SELECT expression FROM table
SELECT column-name, function(column), expression, ... FROM table
```

Column names can be “aliased” to return an alternate name for the column in the header row of the result-set, which is frequently done for expressions.

```
SELECT expression AS column-name FROM table
```

Selection Using WHERE

Selection determines which rows to return in the result-set.

The `WHERE` clause is the primary mechanism used to select result rows; it uses predicate expressions to specify a subset of rows to be returned through boolean comparison operations.

```
SELECT columns FROM table WHERE predicate-expression
```

Basic predicate expressions take on one of the following forms:

```
projection-name operator other-projection-name
projection-name operator literal
literal operator projection-name
```

Where projection-name refers to either a column name, expression, function call, or alias as described in Projection above, and operator is a numeric or string comparison operator from the following list:

Operator	Description
`=`	Equal
`<>` or `!=`	Not equal
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal
`<=`	Less than or equal
`BETWEEN...AND`	Within a range
`LIKE`	Pattern search

Complex predicate expressions are composed of other predicate expressions, and look like this:

```
predicate-expression AND predicate-expression
predicate-expression OR predicate-expression
NOT predicate-expression
```

Complex predicate expressions may be combined with each other, as well, and may contain parentheses in order to define precedence. For example, if you wanted to select all rows with a total count over 1000 that had either a high reject count or high down time, you might use an expression like this:

```
total_count > 1000 and (reject_count > 100 or down_time > 600)
```

Other Selection Types

In addition to the `WHERE` clause, there are other clauses that can be used for filtering rows from result-sets.

The `LIMIT` clause filters rows by limiting the result-set to the specified number of rows; optionally starting at a specified row number specified by `OFFSET` (thus enabling returning result-sets in batches).

The `HAVING` clause is similar to the `WHERE` clause, except that it is applied to a result set after aggregation has occurred, and is only used if a `GROUP BY` clause is present.

Sorting

Sorting orders the rows of the result-set based on any combination of columns.

The `ORDER BY` clause is used for sorting. It is followed by a column to be used for sorting, which can then be followed by `ASC` (the default) or `DESC` to indicate the sort direction. When interacting with an XL device, `ORDER BY` columns must be in the list of columns selected, and, in the Data Schema, must have an

`index` value of either `primary_key` or `yes`.

Aggregation

Aggregation groups and summarizes the result-set based on user-defined criteria.

The `GROUP BY` clause is the primary mechanism for aggregation. When aggregate functions (e.g. `SUM`, `COUNT`, `AVG`, `MIN`, or `MAX`) are included in the `SELECT` clause, `GROUP BY` can be used to return a result-set where the rows contain aggregate data for each `GROUP BY` column value. When interacting with an XL device, `GROUP BY` columns must be in the list of columns selected, and, in the Data Schema, must have an `index` value of either `primary_key` or `yes`.

XL SQL OVER HTTP POST

Request Format

Description
SQL access to the XL device is exposed via a web services API accessed using a key=value pair POST request to the path `/sql-request.do`. The body of the request must specify the following two parameters, in this order, separated by an '&' character:

- `response_type`: Either `application/json`, or `text/xml`; detailed descriptions of the response formats are included below.
- `sql_statement`: A single LESQL statement to be evaluated. The length of the unencoded text cannot exceed 2048 bytes.

The request may be URL encoded, but does not need to be; if it is, the following HTTP header must be attached to the request:

```
Content-Type: application/x-www-form-urlencoded
```

The only other necessary HTTP header is an accurate `Content-Length`.

Example The entire POST request for a statement selecting the most recent 5 sequence numbers and start times from the Timeline Stream might look like this, without encoding:

```
POST /sql-request.do HTTP/1.0
Content-Length: 141

response_type=application/json&sql_statement=select sequence_number, start_time from timeline_stream order by
sequence_number desc limit 5;
```

URL encoded, that same request would look like this:

```
POST /sql-request.do HTTP/1.0
Content-Length: 166
Content-Type: application/x-www-form-urlencoded

response_type=application%2Fjson&sql_statement=select%20sequence_number%2C%20start_time%20from%20timeline_stream%20order%20by%20seq
```

Response Format Common Features

The details of each response format are different, but they all represent the response data as a collection of **Records**, each containing the columns, expressions, and function results requested by the user in the `SELECT` query. Within each Record, the data are organized in the same order as those items specified in the `SELECT` query. (Note that if `*` is used as an argument to `SELECT`, the resulting columns will be in arbitrary order.) No header information is returned; the caller must remember the columns requested and interpret the response data accordingly.

In addition to the returned data, an error field is returned. If there was any error that occurred during processing, it will be presented as a string in this field. It is possible to obtain partial data and an error, so the error field should be checked first. If it is empty, the data is valid; otherwise, no returned data should be used.

JSON Response Format

Description
This section assumes that you are familiar with JavaScript Object Notation (JSON). The XL SQL JSON response format is an array containing a single object that has four fields, as follows:

```
[{
    'data': [ ... ],
    'error': ...,
    'statement_time': ...,
    'total_time': ...
}]
```

The data field contains an array of arrays; each array in the data is a Record as described in *Response Format Common Features*, above.

The `statement_time` and `total_time` fields allow the user to measure the performance of their query. The former measures how much time was spent in the database processing the request, and the latter measures the total time taken by the request. These are both expressed in fractional seconds.

Example
A sample response to the example query described above:

```
[ {
    'data': [
        [ '100473','3481282801.0' ],
        [ '100472','3481275600.0' ],
        [ '100471','3481273800.0' ],
        [ '100470','3481263900.0' ],
        [ '100469','3481263000.0' ]
    ],
    'error': '',
    'statement_time': 0.00440700000000000030,
    'total_time': 0.00475200000000000010
} ]
```

XML Response Format

Description
An XML document is returned, with the following structure:

```
<sql_responses>
```

```
    <sql_response>
        <data>
            <row>
                <cell>...</cell>
                <cell>...</cell>
                ...
            </row>
        </data>
        <error></error>
    </sql_response>
</sql_responses>
```

Each `row` element is a Record as described in *Response Format Common Features* above.

Example
A sample response to the example query described above:

```
<sql_responses>
    <sql_response>
        <data>
            <row>
                <cell>100473</cell>
                <cell>3481282801.0</cell>
            </row>
            <row>
                <cell>100472</cell>
                <cell>3481275600.0</cell>
            </row>
            <row>
                <cell>100471</cell>
                <cell>3481273800.0</cell>
            </row>
            <row>
                <cell>100470</cell>
                <cell>3481263900.0</cell>
            </row>
            <row>
                <cell>100469</cell>
                <cell>3481263000.0</cell>
            </row>
        </data>
        <error></error>
    </sql_response>
</sql_responses>
```

XL SQL DIALECT

Overview

Allowing external access to all of the language features of SQL could result in unbounded resource usage on the XL device (in terms of RAM and processor time) or could cause a loss of data integrity. In order to avoid these issues, the device exposes a subset of SQL called Limited External SQL (LESQL). If full SQL features are needed, the necessary data can be extracted from the XL device and loaded into a fully-featured SQL database for further processing.

In order to preserve data integrity, LESQL only supports the SQL `SELECT` statement. Of particular note within the `SELECT` statement, the following features are disallowed because they might incur unbounded resource use:

- Joins of any sort
- `ORDER BY` on multiple columns, or on any non-indexed column
- `GROUP BY` on multiple columns, or on any non-indexed column.
- The `IN` keyword, and thus sub-`SELECT`s.

This section provides a list of unsupported SQL features, a Backus-Naur grammar for LESQL, and lists of built-in SQL functions available for external access.

Unsupported SQL Features

The following features of SQL are not supported by LESQL:

Language Feature	Type	Reason
`ALTER [ INDEX | TABLE ]`	Command	Database Modification
`CREATE [ INDEX | TABLE | TRIGGER | VIEW ]`	Command	Database Modification
`DELETE`	Command	Database Modification
`DROP [ INDEX | TABLE | TRIGGER | VIEW ]`	Command	Database Modification
`INSERT`	Command	Database Modification
`REPLACE`	Command	Database Modification
`UPDATE`	Command	Database Modification
`[BEGIN | COMMIT | END | ROLLBACK] TRANSACTION`	Command	Unnecessary
`JOIN` operator (and natural join), `UNION, EXCEPT, INTERSECT`	`SELECT` clause	Nondeterministic resource usage
`ORDER BY` (un-indexed argument)	`SELECT` clause	Nondeterministic resource usage
`GROUP BY` (un-indexed argument)	`SELECT` clause	Nondeterministic resource usage
`DISTINCT`	`SELECT` clause	Nondeterministic resource usage
`IN`	`SELECT` clause	Nondeterministic resource usage

Also note that, while `SELECT *` is technically permitted, it is not useful in practice: column names are not returned with column data, and there is no guarantee of the ordering of columns within the data when using `SELECT *`.

LESQL Grammar

Grammar

```
sql-statement ::= SELECT result FROM table
    [ WHERE expr ]
    [ group-by-clause | order-by-clause ]
    [ LIMIT integer [(OFFSET | , ) integer ]]

result ::= result-column [, result-column]*

result-column ::= * | expr [ [AS] string ]

group-by-clause ::= GROUP BY indexed-column-name [ HAVING expr ]

order-by-clause ::= ORDER BY indexed-column-name [ASC|DESC]

expr ::= expr binary-op expr |
    expr [NOT] like-op expr [ESCAPE expr] |
    unary-op expr |
    ( expr ) |
    column-name |
    literal-value |
    function-name ( expr-list | * ) |
    expr [IS | IS NOT | NOT] NULL |
    expr [NOT] BETWEEN expr AND expr |
    CASE [expr] ( WHEN expr THEN expr )+ [ELSE expr] END

like-op ::= LIKE | MATCH | GLOB

unary-op ::= - | + | ! | ~

binary-op ::= <See Binary Operators below>

literal-value ::= <See Literal Values below>

function-name ::= <see Simple SQL Functions and Aggregate SQL Functions below>
```

Binary Operators

SQLite understands the following binary operators, in order from highest to lowest precedence:

```
||
* / %
+ -
<< >> & |
< <= > >=
= == != <>
AND
OR
```

Literal Values

A literal value is an integer number or a floating-point number. Scientific notation is supported. The '.' character is always used as the decimal point. A string constant is formed by enclosing the string in single quotes ('). A single quote within the string can be encoded by putting two single quotes in a row, as in Pascal. C-style escapes using the backslash character are not supported because they are not standard SQL. BLOB literals are string literals containing hexadecimal data and are preceded by a single x or X character. For example:

```
X'53514697465'
```

A literal value can also be the token `NULL`.

Simple SQL Functions

Function	Description
`abs(X)`	Return the absolute value of the numeric argument *X*. Return `NULL` if *X* is `NULL`. Return `0.0` if *X* is not a numeric value.
`length(X)`	Return the string length of *X* in characters if *X* is a string, or in bytes if *X* is a blob.
`lower(X)`	Return a copy of string *X* with all ASCII characters converted to lower case.
`ltrim(X)` `ltrim(X,Y)`	Return a string formed by removing any and all characters that appear in *Y* from the left side of *X*. If the *Y* argument is omitted, spaces are removed.
`max(X,Y,...)`	Return the argument with the maximum value, or return `NULL` if any argument is `NULL`. Note that `max()` is a simple function when it has 2 or more arguments but converts to an aggregate function if given only a single argument.
`min(X,Y,...)`	Return the argument with the minimum value. Note that `min()` is a simple function when it has 2 or more arguments but converts to an aggregate function if given only a single argument.
`nullif(X,Y)`	Return the first argument if the arguments are different, otherwise return `NULL`.
`random()`	Return a pseudo-random integer between `-9223372036854775808` and `+9223372036854775807`.
`randomblob(N)`	Return an *N*-byte blob containing pseudo-random bytes. *N* must be a positive integer.
`replace(X,Y,Z)`	Return a string formed by substituting string *Z* for every occurrence of string *Y* in string *X*. The BINARY collating sequence is used for comparisons. If *Y* is an empty string then return *X* unchanged.
`round(X)` `round(X,Y)`	Round off the number *X* to *Y* digits to the right of the decimal point. If the *Y* argument is omitted, *0* is assumed.
`rtrim(X)` `rtrim(X,Y)`	Return a string formed by removing any and all characters that appear in *Y* from the right side of *X*. If the *Y* argument is omitted, spaces are removed.
`substr(X,Y,Z)` `substr(X,Y)`	Return a substring of input string *X* that begins with the *Y*-th character and which is *Z* characters long. If *Z* is omitted then all characters through the end of the string are returned. The left-most character of *X* is number *1*. If *Y* is negative then the first character of the substring is found by counting from the right rather than the left. If *X* is a string then character indices refer to actual UTF-8 characters. If *X* is a blob then the indices refer to bytes.
`trim(X)` `trim(X,Y)`	Return a string formed by removing any and all characters that appear in *Y* from both ends of *X*. If the *Y* argument is omitted, spaces are removed.
`upper(X)`	Return a copy of string *X* with all ASCII characters converted to upper case.

Aggregate SQL Functions

Name	Requirement
avg(*X*)	Return the average value of all non-NULL *X* within a group. String and BLOB values that do not look like numbers are interpreted as 0. The result of avg() is always a floating point value as long as at there is at least one non-NULL input even if all inputs are integers. The result of avg() is NULL if and only if there are no non-NULL inputs.
count(*X*) count(*)	The first form return a count of the number of times that *X* is not NULL in a group. The second form (with no argument) returns the total number of rows in the group.
max(*X*)	Return the maximum value of all values in the group. The maximum value is the value that would be returned last in an ORDER BY on the same column. NULL is returned if and only if there are no non-NULL values in the group.
min(*X*)	Return the minimum non-NULL value of all values in the group. The minimum value is the first non-NULL value that would appear in an ORDER BY of the column. NULL is only returned if and only if there are no non-NULL values in the group.
sum(*X*) total(*X*)	Return the numeric sum of all non-NULL values in the group. If there are no non-NULL input rows then sum() returns NULL but total() returns 0.0. NULL is not normally a helpful result for the sum of no rows but the SQL standard requires it and most other SQL database engines implement sum() that way so SQLite does it in the same way in order to be compatible. The non-standard total() function is provided as a convenient way to work around this design problem in the SQL language. The result of total() is always a floating point value. The result of sum() is an integer value if all non-NULL inputs are integers. If any input to sum() is neither an integer or a NULL then sum() returns a floating point value which might be an approximation to the true sum. sum() will throw an "integer overflow" exception if all inputs are integers or NULL and an integer overflow occurs at any point during the computation. total() never throws an integer overflow.

XL Specific Functions

Name	Requirement
datetime_to_iso_string(*value [, offset]*)	Convert an XL date_time to the ISO 8601 representation of that date and time. If the offset argument is not provided, the ISO 8601 string will be presented in local time without a UTC offset modifier.
iso_string_to_datetime(*iso_string*)	Converts an ISO 8601 Date/Time string to the value portion of an XL date_time.
vorne_enum_lookup('<table>', '<column>', column_value)	This function may be used to convert an enum value representation to its string representation. String literals containing the names of the table and column to be used as reference must be included in the first two arguments, and the value to be converted is passed as the third argument.