

# Qubik LavaStream Language Reference

# Table of Contents

## Introduction

Manual Scope and Target Audience

Using This Reference

System Requirements

Reference Manual Structure

Language Basics

Program Structuring Mechanisms

Database Interfaces

Text and Structure Interfaces

Programming Environment

LavaStream Syntax

LavaStream Functions

LavaStream Examples

## Language Basics

Abstract

Intent and design of the language

Why not Object Code?

Omission of Pointers

Language Elements

Constants

Basic Types

Assignment

String Manipulation

Concatenation

Declaring a constant

String assignment

Parameter construction and function returns

Substrings (Slicing)

String search

Numeric formatting

XML formatting

Compound Types

Array types

Variables

Typecasting

General Typecasting

Specific Typecasting (Dates)

Specific Typecasting (Times)

Statements

Assignment

String assignment

Assignments to structures

Assignments from XML data into a structure

Assignment into structures from Lava object row types

Assignments between structures

Element assignment

[Assignment between equivalent structures](#)  
[Assignment between non-equivalent structures](#)

[Conditions](#)  
[Case statements](#)  
[Loops](#)  
[Procedure calls](#)  
[Return statement](#)

[Formal Procedure Declaration](#)  
[Comments](#)  
[External Function Calls](#)

## [Built-in Functions](#)

[Date and Time functions](#)  
[Variable functions](#)  
[Lava Database functions](#)  
[ODBC Database functions](#)  
[String Functions](#)  
[Text output and formatting functions](#)  
[Text file functions](#)  
[Windows related functions](#)

## [Program Structuring Mechanisms](#)

[Procedures](#)  
[Procedure Declaration](#)  
[Functions](#)  
[Modules](#)  
[Projects](#)  
[Import Mechanism](#)  
[Inter-module Imports](#)  
[Inter-project Imports](#)

## [Database Interfaces](#)

[Lava Server Interface](#)  
[SQL Interface](#)  
[Row-level Interface](#)  
[Row-Level Seek](#)  
[Lava Interface Selection](#)  
[Error checking](#)  
[ODBC Interface](#)  
[Connecting to ODBC database servers](#)  
[SQL Interface](#)  
[Result Row Access](#)  
[Row Insertion](#)  
[Row Updates](#)  
[Closing ODBC connections](#)

## [Text File and Structure Interfaces](#)

[Text File Interface](#)  
[Writing Text Files](#)  
[Reading Text Files](#)  
[Completing File I/O](#)

## XML Interface

[XML Import](#)

[XML Export](#)

## Programming Environment

[Blueprint Operating Principles](#)

[Programming using Qubik Blueprint](#)

[Graphical Object Management](#)

[Creating Nodes](#)

[LavaStream Projects](#)

[LavaStream Modules](#)

[Editing and Compiling LavaStream Modules](#)

[Module List](#)

[Procedure List](#)

[Compile Button](#)

[Dictionary Definition](#)

[Table Definition](#)

[Column Definition](#)

[Definition of Relations](#)

[Generating a Dictionary](#)

[Creating Data Tables on a Lava Server](#)

[Integrating LavaStream with a Dictionary Schema](#)

[Testing using LavaStream Elements](#)

[Adjusting the Elements Mount Parameters](#)

[Running in Exclusive Mount](#)

[Executing a LavaStream Procedure](#)

[Selecting a LavaStream Procedure for execution](#)

[Viewing Output](#)

[Debugging LavaStream](#)

[Examine Variables](#)

[Variables](#)

[Arrays](#)

[Structures](#)

[Setting Breakpoints](#)

[Unconditional](#)

[Disabled](#)

[Conditional](#)

[Executing the Code](#)

[Step Over - F10](#)

[Step Into - F11](#)

[Step Out](#)

[Go to Breakpoint - F5](#)

[Viewing Output](#)

[Exit the Debugger](#)

## LavaStream Syntax

[Formal Syntax](#)

## LavaStream Functions

[Date and Time functions](#)

[Date](#)

[DateFormat](#)  
[Day](#)  
[Hour](#)  
[Minute](#)  
[Month](#)  
[Second](#)  
[TimeFormat](#)  
[Year](#)  
[Time](#)  
[Datetime](#)  
[MsDate](#)

### Variable functions

[BITAND](#)  
[BITOR](#)  
[BITXOR](#)  
[Clear](#)  
[Dec](#)  
[Even](#)  
[Inc](#)  
[Entier](#)  
[Length](#)  
[Odd](#)  
[Round](#)  
[Abs](#)

### String functions

[Char](#)  
[IPfromString](#)  
[Lower](#)  
[Ord](#)  
[ReplaceChar](#)  
[Slicing](#)  
[StringFromIP](#)  
[StringPos](#)  
[TrimAll](#)  
[TrimLeft](#)  
[TrimRight](#)  
[Upper](#)  
[Length](#)

### Lava Database functions

[AutoCommit](#)  
[Clear](#)  
[ColumnLabel](#)  
[ColumnLength](#)  
[Commit](#)  
[Createtable](#)  
[DropIndex](#)  
[Droptable](#)  
[FindClose](#)  
[FindColumn](#)  
[FindNext](#)  
[FindRow](#)  
[FindTable](#)  
[FreeRow](#)  
[GetColumn](#)

[LavaError](#)  
[LavaStatus](#)  
[OpenSession](#)  
[RenameTable](#)  
[ReserveRows](#)  
[Rollback](#)  
[SetSession](#)  
[Session](#)  
[SetColumn](#)  
[Sql](#)  
[Tablecolumns](#)  
[TableName](#)  
[Tablerows](#)  
[Translate](#)  
[Truncate](#)

### [ODBC Database functions](#)

[ODBC Close](#)  
[ODBC Connect](#)  
[ODBC Sql](#)  
[ODBC Add](#)  
[ODBC Update](#)

### [Text Output and Formatting Functions](#)

[Output](#)  
[Format](#)  
[XmlHeader](#)  
[XmlRowdata](#)  
[XmlFooter](#)

### [Text file functions](#)

[Createfile](#)  
[Writefile](#)  
[WritefileNL](#)  
[Readfile](#)  
[Spacefile](#)  
[Closefile](#)  
[Appendfile](#)  
[Setfilepos](#)  
[Setpath](#)

### [Generic file functions](#)

[FindFile](#)  
[NextFile](#)

[FindFolder](#)  
[NextFolder](#)  
[WinFileCopy](#)  
[WinFileDelete](#)  
[WinFileMove](#)  
[WinFileRename](#)  
[ReadMemFile](#)  
[FreeMemFile](#)  
[ServerFetchFile](#)  
[ServerPutFile](#)  
[WriteMemFile](#)

### [Window Related Functions](#)

[Message](#)

## [Miscellaneous functions](#)

[Exec](#)

[WinFormatMessage](#)

[WinGetLastError](#)

[CreateNode](#)

[NodePath](#)

[ResolvePath](#)

[Sleep](#)

[SendMail](#)

[SetDecimals](#)

[Thread](#)

[WriteLog](#)

## [Apache Configuration and Testing](#)

### [Installing Apache](#)

[Installing under Windows XP or Windows Server 2003 \(Windows 5.1 / 5.2\)](#)

[Installing under Windows Vista](#)

### [Apache Configuration](#)

[Using the Qubik Apache Install](#)

### [Manual Apache Configuration](#)

[Apache Configuration File](#)

[Lava Apache Module](#)

[Lava ODBC Driver](#)

[Lava Javascript Library](#)

### [Setting up the Apache Lava Client](#)

[Logging](#)

[LogLevel](#)

[Server](#)

[ClientPath](#)

[User](#)

[Password](#)

### [Testing with Apache](#)

## [FormatX - Styles and Attributes](#)

## [LavaStream Examples](#)

### [Coding Techniques](#)

[Very Simple Program](#)

[Temporary Table as Array Replacement](#)

### [Windows Programming](#)

[Message Box](#)

### [Lava Database Access](#)

[Simple Loop on Lava Table](#)

### [ODBC Database Access](#)

[Retrieving ODBC Data](#)

### [XML Interfacing](#)

[Simple XML Output](#)

[Basic XML input](#)

[XML List Output](#)



# List of Illustrations

Blueprint Desktop (-38-)  
Node Selection Button (-39-)  
Blueprint Editor (-40-)  
Dictionary Example (-43-)  
Dictionary Table Properties (-44-)  
Blueprint Relation Tool (-44-)  
Table Relation Properties (-45-)  
Invoking the Debugger (-49-)  
LavaStream Debugger (-49-)  
Debugger Breakpoints (-50-)  
Debugging Output (-51-)  
LavaStream Elements Interface (-120-)

# Introduction

## Manual Scope and Target Audience

This reference manual covers the usage of and programming using the LavaStream programming language.

**This manual is targeted at serious and casual programmers and database users who wish to do one or more of the following :**

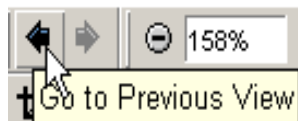
- Write WebServer routines to satisfy Web based applications using a Lava Server database through the WebLava interface
- Write data porting applications to retrieve information from an ODBC database source and place potentially translated data in a Lava Server database
- Write, modify or enhance dictionary template procedures for generating data dictionary related routines or program modules
- Write, modify or enhance window designer procedures for generating form-based routines or program modules
- Create or maintain stored procedures used in a Lava Server or Satellite database

**Topics covered include :**

- Language familiarization
- Programming principles
- Basic use of programming environments (Qubik Blueprint and LavaStream Elements)
- External interfaces (ODBC and XML)
- Language Syntax

## Using This Reference

This PDF document is extensively hyperlinked. In order to derive the best usage from this document, note that the Acrobat PDF reader has a “back” button (just like a browser) located in the toolbar, which looks like this :



After following a hyperlink (any blue underlined text) by left-clicking on the hyperlink with your mouse, clicking the “back” button will return you to the hyperlink just accessed. The “back” button will work through multiple levels of links.

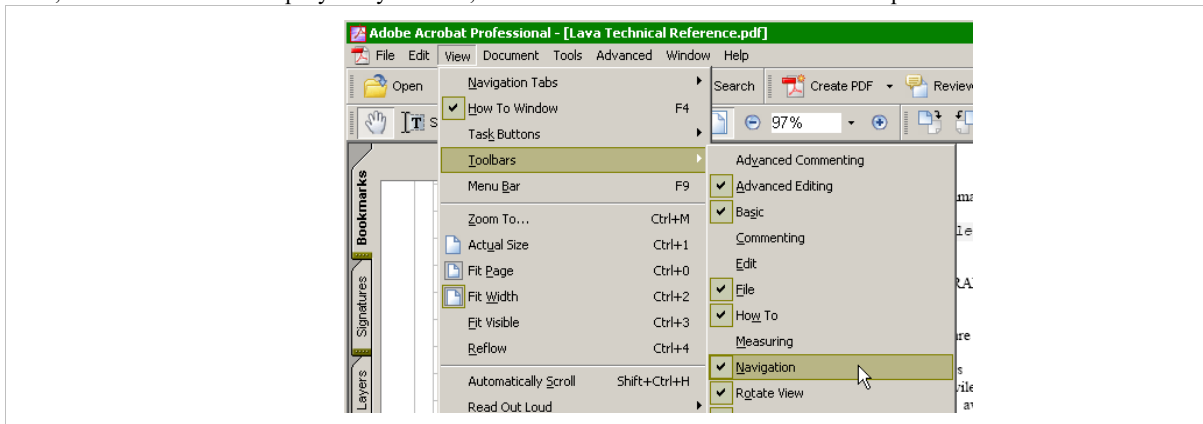
The “back” option can also be found in the mouse right-click pop-up menu, named “Go to Previous View”.

Note that as of Acrobat 6, the “back” option is no longer presented in the pop-up menu, and the toolbar looks slightly different :



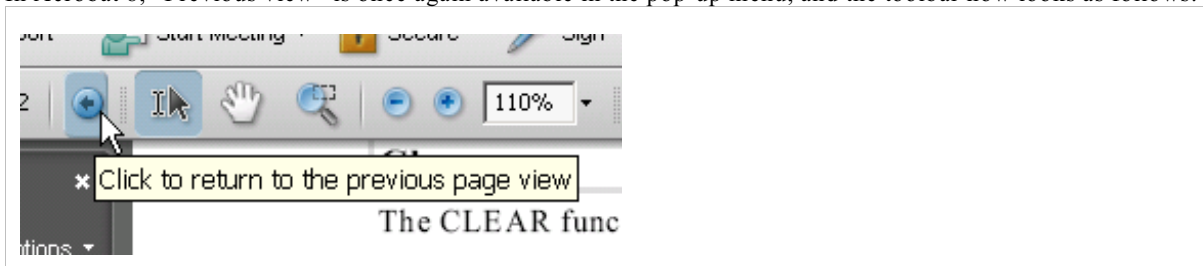
## Manual Scope and Target Audience

Also, the toolbar is not displayed by default, and must be activated in the Toolbar options as follows :



There is a keyboard shortcut for the *Previous View* function, which is *Alt+Left Arrow*

In Acrobat 8, “Previous view” is once again available in the pop-up menu, and the toolbar now looks as follows:



To enable this toolbar option, go to “More tools” in the pop-up menu when right-clicking in the toolbar area, and select “Previous View” from the Page Navigation Toolbar.

## System Requirements

LavaStream requires access to a Qubik Lava Server database for operation. The LavaStream Engine runs on any Windows workstation or server release from Windows XP / Windows 2003 onward. Browser-based applications can be run on any workstation supporting Firefox (revision 2 or later), or Internet Explorer (revision 6 or later). Apache 2.2 or later is the only supported Web server.

## Reference Manual Structure

The manual is divided into a number of sections, each of which addresses a specific topic.

### Language Basics

A description of the syntax and semantics of the LavaStream language. Information is provided on the design and implementation of the LavaStream system.

### Program Structuring Mechanisms

Information on the hierarchical structure of a LavaStream project, including project, module and procedure mechanisms

### Database Interfaces

## Manual Scope and Target Audience

A description of the integration of LavaStream with the Lava database as well as information on the ODBC connectivity presented in LavaStream.

### **Text and Structure Interfaces**

Specification of the available techniques for constructing textual and XML output through the LavaStream text functions

### **Programming Environment**

A brief description of the Blueprint development environment. For detailed information see the Qubik Blueprint Operation Guide.

### **LavaStream Syntax**

A detailed description of the syntax of the LavaStream language, including a formal definition of the LavaStream syntax

### **LavaStream Functions**

Specification and examples of the built-in functions which extend LavaStream for use in database and browser environments, amongst others.

### **LavaStream Examples**

A set of worked programming examples which illustrate the use of several LavaStream techniques and functions.

## Language Basics

This chapter addresses the basic aspects of the LavaStream language. Programmers not familiar with Pascal, Modula-2 or Oberon should read through this chapter to gain familiarity with the structure of the language.

### Abstract

LavaStream is an interpreted language tightly integrated with the Lava Database.

Close integration with the Apache Web server allows LavaStream programs to be triggered from a Web browser client to respond to complex data-driven page formatting and data output requests.

As LavaStream programs are stored in the Lava database, they may be used as stored procedures and may be triggered remotely or locally to perform complex data manipulation or validation tasks.

Due to integrated ODBC functionality, LavaStream may be used to extract data from third-party databases, and potentially to reformat, process and re-insert data, potentially into a different database than the originating data. Using the automatic distribution facilities of the Lava database coupled with the ability to run LavaStream programs on Lava Satellite servers, data may be collected, stored, processed and inserted to or from databases residing on different continents, if required.

Coupled to the ability of Lava Satellite servers to link to a Lava Primary server across an internet link, LavaStream presents the ability to run browser-based thin client applications on remote sites using their own Apache Web server, thus attaining local area network (LAN) speeds in an application which stores and retrieves its centralised data on a remote site. This allows fast response and processing as well as bandwidth independence while retaining centralised data storage.

As LavaStream programs are automatically distributed across a Lava database network to all Lava Satellite servers, LavaStream program updates performed at the central server automatically deploy to all satellite sites within seconds. This eliminates complex installation and update procedures, as all linked sites immediately upgrade to the latest revision of the software.

LavaStream is engineered to be extremely fast and efficient. Even a moderate Web server running Apache and LavaStream will be capable of serving thousands of browser clients - due to the tight integration of LavaStream with the Lava database, and the distributed architecture used in the Lava Server and the LavaStream client, typical requests from browser client applications are serviced within a few milliseconds, allowing very large numbers of browser clients to be served without processor or bandwidth limits being reached. Coupled to the ability to decentralize servers using a Lava Satellite network, it becomes possible to configure business application networks serving tens of thousands of browser clients with almost insignificant hardware cost.

When using a Lava Satellite network with localised Apache Web servers to decentralize client load, the bandwidth load on the network to the central Lava and Apache servers is decreased dramatically. Transaction loads running into hundreds of transactions per second can be processed with a total bandwidth demand from decentralised Satellite servers of no more than 64 kilobit per second, resulting in the available bandwidth to the central Web server being freed up for other purposes.

### Intent and design of the language

LavaStream is intended to provide a powerful, easy to use interface between the Lava database and external systems. Specifically, interfaces are provided to Apache for Internet interface, ODBC interface for access to third-party databases, text file interface for creation of text files, and XML interface for import and export of XML structure sets.

## Manual Scope and Target Audience

Extensive and highly functional programs can be written using LavaStream, but it must be understood that LavaStream is not intended to be a completely general-purpose programming language. Although it is certainly possible to perform a very wide range of tasks using LavaStream, the primary intent of the system is to provide an interface to databases, and most specifically to the Lava Server database.

The primary targets of LavaStream are :

- Programming WebServer-based systems using a browser as the client interface
- Writing or extending code generation routines for the Blueprint Dictionary system
- Writing or extending code generation routines for the Blueprint Browser Window system
- Writing stored procedures for a Lava Server database

The design of the language is aimed at providing a means of writing programs in a way which allows the programmer the maximum in flexibility while retaining structural integrity, and ensuring that the language is legible and maintainable.

Basic language syntax is patterned on Oberon, a language very similar to Pascal in syntax. The choice of Oberon as a starting point is due in part to the small language definition, providing a very quick learning curve, and the high legibility of Oberon programs, resulting in easy maintenance.

A few modifications to the Oberon language have been adopted.

1. Only linear (non-object) code is supported. This allows the language - and therefore the runtime - to be simpler, and hence also smaller and faster. Although object code does have advantages in certain forms of applications, this is more than offset by the tight integration of LavaStream with the Lava database, which presents a functional enhancement in some cases greater than would have been provided through object code.
2. Automatic typecasting is supported. This violation of the strict typecasting in Oberon was decided due to the demands for rapid development of software, and the corresponding easing of requirements on the programmer to achieve a given aim.
3. Some variable types were eliminated. Since LavaStream is a special-purpose language, several variable types and constructs (such as unions, for example) are not considered mandatory. In general, however, all the important data types and structures are supported.
4. Return of compound types was included. This renders several complex operations possible with greater ease than in Oberon.
5. Procedure-level declaration of constants and types is disallowed. There is little benefit to procedure-level scoped constants and types, with significant extra complication in both compiler and runtime engine, and thus it was decided to eliminate this option.
6. Nested procedures are not supported. This would complicate the runtime engine to a significant degree without a comparable degree of improvement in functional scope.

## Why not Object Code?

We would not argue against the statement that there is a place for object code in certain applications. However, in many (if not most) applications where object code is appropriate or valuable, the most important function served by the class / object mechanism is the storage and retrieval of runtime data relating to the operation of the software. The actual method usage in many cases is subservient to the convenience of having the built-in object linkage provide for transient storage of data (in most environments lost on program exit) allowing complex linked structures to be constructed, traversed and retrieved during execution.

In the LavaStream environment, the designer / programmer has an entire database at their disposal, including the ability to create any number of virtual tables (defined and accessed entirely in memory) with very little effort - certainly no more effort than that required to define and construct a class / object system. As a result, the requirement for storing transient data is fully addressed without the requirement for objects, while the virtual table mechanism is both faster and more versatile than a linked object implementation. In addition, the ability to

## Manual Scope and Target Audience

define permanent storage just as easily allows vastly more functionality than provided by a object system. In a LavaStream environment, due to the distributed nature of the Lava database, physical (permanent) tables operate just as fast as virtual tables under the majority of circumstances, giving the best of all scenarios.

The only omission, therefore, is the method mechanism - which is only really of greater value than a conventional function call when overloading allows different methods to be selected depending on the type of the object processed. It is our viewpoint that this practice - powerful as it may be - is adverse to good code maintenance, as it obfuscates the functionality of the code by rendering the actual function executed through the method non-obvious. For this reason, this sole advantage of object systems over the LavaStream implementation is seen not as an advantage at all, but rather as a maintenance detriment. As programs are written once and maintained many times, the maintenance disadvantage outweighs the functionality represented by overloading very easily. Given this deduction, it was an easy and logical decision to eliminate class / object functionality entirely in order to improve the overall maintainability of the system.

In summary, the true advantages of object systems, primarily represented by transient data storage, are easily compensated through the direct integration of LavaStream with the Lava database. Inheritance and overloading are viewed as of dubious value in the LavaStream domain (that being database and human interface centric applications) and are therefore discounted. The advantages of a small, fast, reliable runtime system as implemented in LavaStream outweigh any residual positive aspects of an object system easily. For comparison, the Sun Java runtime is more than 6 Mb in size. The LavaStream runtime is approximately one tenth of that size, while the effective functionality presented by the two systems - although fairly different - is very similar in total scope. As a second example, the PHP runtime is over 12 Mb in size, and the total functionality presented in PHP is significantly less than that included in LavaStream, while LavaStream execution (in exactly the same domain) is significantly faster.

## Omission of Pointers

The specification and use of pointers and pointer-type variables has intentionally been omitted in LavaStream. Although there is no doubt that pointers are extremely flexible, and provide a very powerful mechanism for abstraction when used correctly and carefully, it is equally true that pointers provide many possibilities for memory corruption and access violation.

In terms of the requirement to provide a run-time environment which is as stable and reliable as possible, while also providing the programmer with a language which promotes stable and reliable code, it is our view that providing pointers in this environment is generally more likely to lead to instability than is the case with other variable types and reference mechanisms.

For this reason, the LavaStream language is defined in such a way that almost any desired result may be obtained without the use of pointers. Procedure parameter declarations provide a by-reference syntax (equivalent to that used in Pascal or Oberon) as follows :

```
VAR parameter : INTEGER;
```

In the above parameter specification, the VAR keyword specifies that the parameter is by reference, in other words modification of the parameter in the called procedure will result in equivalent modification of the nominated actual variable in the calling procedure. The system provides this by reference link without resorting to the use of pointers, and it is not possible to cause memory corruption in the called procedure through any manipulation of the parameter.

Where a particular result is desired which cannot be envisaged without the use of pointers, the advanced programmer may simply resort to calling an externally defined procedure.

### See also

Formal Procedure Declaration, External Function Calls

## Language Elements

This section describes the basic elements of the LavaStream language. For further information on use and implementation of LavaStream code, see the syntax and examples sections.

### Constants

Constants are supported only at module level - procedure constants are not provided.

As with variables, constants permit mixing of types - the result of a constant construction will be a value of the most sensible type given the nature of the constant definition.

Constant definitions such as those following are supported :

```
PI           =      3.141592E00;
Name        =      'Jones' ;
```

Complex constants may be constructed :

```
Code        =      3.5 * 2 / 8.7;
Password    =      'aBc' & 22/7 & 'xag' ;
```

Character codes defined in hex may be used :

```
NEWLINE     =      0DX & 0AX;
Line        =      'Start' & 09X & 'after tab character' ;
```

### Basic Types

LavaStream supports a sufficiently diverse set of basic variable types to allow programming of most algorithmic and data processing requirements.

DATE	A julian date representation, with the earliest being 1 Jan 300 CE
TIME	A numeric representation of time with 0 representing midnight, and a non-zero number representing the count of milliseconds since midnight
DATETIME	A combined date and time representation, with the integer portion being the julian date, and the fractional portion being a count of milliseconds since midnight, divided by 100,000,000
BOOLEAN	A byte-sized variable with allowable values TRUE and FALSE only. Internally, TRUE is equivalent to 1 and FALSE is equivalent to 0
BYTE	An unsigned byte-wide variable, which may contain one character or a numeric unsigned value of maximum 255
SHORT	A signed, 2-byte integer
INTEGER	A signed, 4-byte integer
IP	Equivalent to a 4-byte integer, but specifically intended to store IP (internet protocol) addresses.
QUAD	An unsigned, 8-byte integer
SHORTFLOAT	A 4-byte floating point number
FLOAT	An 8-byte floating point number
STRING	A string variable. The length of the variable is generally unspecified, with the exception of ROWTYPE compound variables (see below). The variable will automatically extend to any size to accommodate required input
TABLE	A variable intended to store table identifiers, for example resulting from SQL select statements

	(see Lava functions below)
ROWTYPE	A compound type (structure) - see below
LAVA	A keyword used to define Lava Server table structures - see compound types below

### Assignment

Assignment between variables or from a function procedure to a variable is accomplished through the same syntax as with Pascal or Oberon :

```
Var1 := Var2;
```

Most variables may be assigned to one another, with the exception that complex variables (such as arrays or structures) may not be assigned to simple variables (such as integers or floats) or vice versa. A partial exception to the first rule is that strings may be assigned to simple types, and the result is the interpretation of the content of the string in terms of the type of the target variable. For example, assigning a string which contains the value "12.34" into a float variable will result in the float variable containing the float value 12.34. Similar appropriate results will be produced with assignment into integers or booleans. Simple types may also be assigned into strings, with the resultant string value being the closest match to the variable value (for example, assigning a boolean into a string will have the results "TRUE" or "FALSE").

In general, arrays of equivalent type may be assigned to one another (regardless of specific size issues) and structures which are sufficiently equivalent to produce sensible assignment may be assigned - the LavaStream runtime will produce the best possible results given the most applicable match between source and target variables.

Assignment of function procedure results into a variable (either built-in functions or user declared) operates on the same principle as variable assignment, for example :

```
DateVar := DATE();
```

The casting rules are identical to those for variables, including functions which return string variables.

#### See also

[Typecasting](#)

### String Manipulation

Due to the requirements for complex string processing, especially in the Web domain, strings are given special attention in LavaStream. The following facilities are supported :

#### Concatenation

Concatenation is supported in every possible syntactical point where a string may be used. The following examples illustrate usage of concatenation

##### *Declaring a constant*

```
Password = 'aBc' & (34 / 7) & 'xxx';
```

Note in the above example the brackets around the calculation, 34 / 7. Without these brackets, the concatenation operator (&) will take precedence over the division (/) and will execute first, yielding "aBc34". The division will execute afterward, resulting in 0.0 (since the first string will evaluate to numeric 0).

##### *String assignment*

```
StringVar := StringVar & 23.33 & (FloatVar / 2) & StringVar2;
```

##### *Parameter construction and function returns*

```
StringVar := FuncProc1(StringVar & 'abc' & FloatVar) & 'endofstring';
```

### Substrings (Slicing)

The most efficient and unambiguous way to extract substrings from a string variable is through a technique known as slicing. This is illustrated in the following example

If StringVar contains the string 'abcdefgh', the following slice instruction

```
NewSubString := StringVar[3..5];
```

yields the string 'def', whereas

```
NewSubString := StringVar[0..3];
```

yields the string 'abcd'.

Variables or any other form of numeric derivation may be used as parameters in the slice, for example

```
NewSubString := StringVar[IntegerVar1..IntegerVar2];
```

### String search

String searching is supported through the function STRINGPOS, as illustrated below

```
Charpos := STRINGPOS('abcdefgh', 'bc');
```

which yields the result 1 - position 0 is the first character in the string.

The STRINGPOS function may be used in an alternative mode, when the starting position of the search must be stipulated.

```
Charpos := STRINGPOS('abcabcdefgh', 'bc', 2);
```

which yields the result 4, as the starting position of the search is position 2, thereby eliminating the first occurrence of 'bc' as a result.

### Numeric formatting

Several quantities are represented numerically in LavaStream, such as dates (in Julian form) and times (in milliseconds since midnight), therefore requiring formatting into string form for output purposes. To support this, the function FORMAT is provided, which allows various numerical representations to be converted to corresponding string representations. Some examples follow - for more detailed information on the FORMAT procedure, see the LavaStream Syntax chapter.

CONST

```
DateStyle = FormatX.FORMAT_P_DATE;
```

```
DateFormat = FormatX.FORMAT_S_DATE_9; (* selected date format is yyyy/mm/dd *)
```

```
TimeStyle = FormatX.FORMAT_P_TIME;
```

```
TimeFormat = FormatX.FORMAT_S_TIME_2; (* selected time format is HH.mm.ss *)
```

```
OUTPUT(FORMAT( DATE(), DateStyle, DateFormat));
```

Built-in function DATE returns the current date

```
OUTPUT(FORMAT( TIME(), TimeStyle, TimeFormat));
```

Built-in function TIME returns the current time.

Many other possibilities exist, ranging from a variety of date and time formats (listed in the supplied module FormatX, which specifies the constants equating to various formats and styles) through numeric formats such as currency and percentage, through floating point formats.

### XML formatting

In order to support XML interfaces, both to client-side JavaScript as well as other external interfaces, LavaStream supports a set of XML output formatting procedures. An example is provided below.

The following type definition provides a structure for XML output :

```
TYPE
  FormType = ROWTYPE
```

```

customer    :   STRING[100];
address     :   STRING[100];
country     :   STRING[100];
END;
```

The code below will construct XML output in the text stream :

```

Form.customer := 'John Smith';
Form.address := '238 Stream Street, London';
Form.country := 'England';
Form.ROWID := 234; (* optional *)
OUTPUT(XMLHEADER(TRUE, 'items'));
OUTPUT(XMLROWDATA('formdata', Form));
OUTPUT(XMLFOOTER('items'));
```

In the above code, the TRUE specification in the XMLHEADER parameters stipulates that the ?xml version specification is output in the header.

The following XML will be produced :

```

<?xml version="1.0" encoding="utf-8" ?>
<items>
  <formdata ROWID="234">
    <customer>John Smith</customer>
    <address>238 Stream Street, London</address>
    <country>England</country>
  </formdata>
</items>
```

Note the ROWID attribute in the above XML - this is produced to allow interface to client forms, so that the database row information is preserved. If the ROWID attribute to the form is zero, this information is omitted.

#### See also

[Specific Typecasting \(Dates\)](#), [Assignments to structures](#)

## Compound Types

LavaStream supports two forms of compound structure types.

The first is used to define structure types compliant with Lava Server tables (either user tables or system tables to which sufficient access privilege exists).

This definition takes the form

```

TableVar    :   LAVA.EVENT.Sys_Event.TYPE;
```

In this definition, the prefix LAVA qualifies the type as a lava table definition. The EVENT qualifier specifies the Event Lava database schema. The identifier Sys\_Event qualifies the Sys\_Event table in the Event schema. The suffix TYPE specifies that the variable acquires the row type of this table.

The second compound type is used for the definition of user-defined row types. An example follows

```

TYPE
  FormType  =   ROWTYPE
  customer  :   STRING[50];
  address   :   STRING[100];
  ID        :   INTEGER;
  district  :   STRING[60];
END;
```

## Manual Scope and Target Audience

The preceding definition in the TYPE section defines a compound type, called FormType. This specifies a row type comprising three string variables of specified length (50, 100 and 60 bytes respectively) and a 4-byte integer field. Note that in contrast with basic string variables (which are specified without length) string variables in row types are specified with a given length. This is necessary as row types may be used to create tables in the database, and specific length of fields defining table columns is therefore required.

Row types may be used to define variables or parameters, as follows

```
FormData          : FormType;
```

Rules of assignment of row types are defined as dictated by logic.

- Compound variables of differing row types may be assigned to one another provided the number of fields in each row type is the same. Fields are typecast on a field for field basis.
- Table rows may be assigned from a Lava data table into a row type variable provided the number of fields in the row type is the same as the number of columns in the table. Fields are typecast on a column-to-field basis.
- Variables of a given row type may be assigned into a row of a Lava data table provided that the number of fields in the row type is the same as the number of columns in the table. Fields are typecast on a field-to-column basis.

### Array types

#### Arrays of simple types

Arrays of several simple types are fully supported in LavaStream. These include arrays of String, Boolean, Integer, Float and Quad values. Arrays of character are supported through the STRING built-in type. Only single-dimensional arrays are supported in the current release.

Arrays in LavaStream are indexed from index 0 (0-based).

In order to present the most usable definition of an array variable, LavaStream does not require the specification of an array bound. The following example illustrates the definition of an array :

```
VAR  
ArrayVar          : ARRAY OF INTEGER;
```

In the case of string arrays, the length of the string must be specified, as in the following example :

```
ArrayVar2         : ARRAY OF STRING[100];
```

Note that in this example, no bound (limit) to the array size is specified. LavaStream will allocate memory as required, to a limit of 100,000 elements. The array will naturally resize as required, whenever an element is written into the array as follows :

```
ArrayVar[230] := 99;
```

In the above example, regardless of the previous size of the array, the assignment as specified will automatically enlarge the array to 230 elements (actually 231, since element 0 is a valid array element).

Although reading from a nonexistent element in an array will not cause run-time failure, it will return 0.

Arrays may be passed to procedures as parameters both using by reference (VAR) or by value parameters. Arrays may also be assigned to other array variables, provided that the type of the array (Integer, Boolean..) is

## Manual Scope and Target Audience

the same for both array variables. In the case of string array variables, the length of the string must also be the same.

Arrays may also be returned as the return type of a function-type procedure. The same provisions apply to returned arrays as for other array assignments.

### Arrays of complex types

Arrays of structures are not directly supported in LavaStream in a conventional programming sense. In order to simultaneously reduce the complexity of the language and eliminate the problems associated with array size limits, LavaStream uses tight integration with the Lava database to support infinitely extensible arrays in the form of Lava data tables (virtual or physical). These are addressed - both for writing and reading - with conventional array syntax, and writes to arbitrary array indices are supported through automatic extension of the data table.

See the example [Temporary Table as Array Replacement](#) for an illustration of table creation and access.

## Variables

Both module-level and procedure-level variables are provided for. Conventional rules of scope apply to visibility and persistence of variables.

Variables may be declared of any basic variable type, or of user-defined row types. Variables may also be declared of Lava Table types, which result in a variable of the same type as the nominated table column definition.

Basic data types behave as per conventional rules for fixed-length variables. However, string type variables in LavaStream behave differently than those in conventional formal languages. String variables are automatically extended to accept assignment of arbitrary length, in order to accommodate lengthy, complex composition of (for example) web pages. Arrays of simple types behave in essence the same as string variables - assignment to a given index in the array ensures that the array is at least of sufficient size to store the nominated index.

## Typecasting

### General Typecasting

In order to provide a programming environment which allows sufficient freedom to code without impediment, any sensible form of typecasting between variable types is supported implicitly. In other words, even variables such as strings and integers may be assigned to one another freely.

It is understood that in a weakly typed language such as this, certain forms of type violations which may constitute programming errors would escape detection. In programming, much of design is based on compromise, and given the intent of this language it was felt that a weakly typed model was the more appropriate one.

For example, the following line of code

```
OUTPUT ('<td>' & FormData.InvoiceTotal & '</td>');
```

would be considerably less convenient to code if automatic typecasting between a numeric value (FormData.InvoiceTotal) and the string type of the parameter to OUTPUT() were not permitted.

Thus, all typecasting which can be sensibly deduced is supported without warning or error. It is, therefore, the

## Manual Scope and Target Audience

responsibility of the programmer to ensure that the results of such variable typecasting is desirable.

In the same manner as the above, automatic typecasting is supported between a number of basic types :

Target Type	Permissible Source Types	Comments
INTEGER	BOOLEAN FLOAT INTEGER QUAD STRING	Where the target type does not provide for the capacity of the source type, truncation or elimination of detail may occur.  String assignment into numerical types requires a valid numerical formatted string for successful conversion.
FLOAT	BOOLEAN FLOAT INTEGER QUAD STRING	Same constraints as above
BOOLEAN	BOOLEAN FLOAT INTEGER QUAD STRING	A boolean target interprets the source as TRUE if numerically non-0, and FALSE if numerically 0.  For a string source, the strings TRUE and FALSE are specifically tested.
QUAD	BOOLEAN FLOAT INTEGER QUAD STRING	Assignment into Quad variables typically addresses only the Low part of the variable, with the exception of very large float values and other Quad variables.
STRING	BOOLEAN FLOAT INTEGER QUAD STRING	Conversion from boolean value to string yields the strings TRUE or FALSE as result.

Note that in the above table only root types have been listed. Implicitly, if conversion is supported between INTEGER and FLOAT, then conversion is also supported between SHORT and SHORTFLOAT. In cases where the assignment target is smaller than the source, the possibility of overflow exists which will cause the target variable in the assignment to have an invalid or unrelated value after the assignment has been performed.

### Specific Typecasting (Dates)

In order to allow conversion from textual dates to Julian dates, the system supports assignment from strings into variables of type DATE with analysis of the string to deduce the date implied.

In order to provide for different source date formats, the built-in function DATEFORMAT is provided, which accepts the following date formats :

FORMAT_S_DATE_1	- = 1;	(* dd/mm/yy *)
FORMAT_S_DATE_2	- = 2;	(* dd/mm/yyyy *)
FORMAT_S_DATE_3	- = 3;	(* mm/dd/yy *)
FORMAT_S_DATE_4	- = 4;	(* mm/dd/yyyy *)
FORMAT_S_DATE_9	- = 9;	(* yyyy/mm/dd *)

Once a default date format has been set, if a string or string variable is assigned into a DATE- type variable, the string is interpreted as a date in the given format. The date is converted into a Julian-format date and assigned into the date variable.

### Specific Typecasting (Times)

tbd

## Statements

The following programming statements are supported in LavaStream

### Assignment

Assignment in LavaStream, as in Pascal or Oberon, uses a more formal assignment symbol in order to distinguish assignment from equality comparison ( $A = B$ ).

```
Var1 := Var2;
```

Assignment into defined variables is supported from :

- Other variables either of the same type or with allowable typecasting
  - Functions returning an appropriate type
  - Rows from Lava tables where the variable is of an appropriate row type or Lava row type
- Automatic typecasting on a variable or field basis is applied on assignment.

### String assignment

String assignment is extended in several ways, given the particular requirements of string variables.

Firstly, concatenation is supported during assignment :

```
StringA := StringB & 'abc' & StringC;
```

In the above assignment, both a literal string ('abc') and a variable (StringC) are concatenated to the base assignment (StringB).

Secondly, automatic type conversion is performed if the source variable is not a string. In the example below, if VarA is a float variable containing the value 23.567, the assignment

```
StringA := VarA;
```

will result in the string value '23.567' in the variable StringA.

Thirdly, string slicing is supported in both the left and right-hand of the assignment :

Given the following starting values for two variables :

```
StringA      :      '0123456789'  
StringB      :      'ABCDEFGH'
```

The assignments as below would yield results as stipulated :

```
StringA := StringB[2..4];
```

'CDE'

```
StringA[3..5] := StringB;
'012ABCDEFG6789'
```

Note : The entire StringB is inserted into StringA, replacing segment 3..5

```
StringA[3..4] := StringB[0..3];
'012ABCD6789'
```

Also :

```
StringA[4] := StringB[2];
'0123C56789'
```

```
StringA[4] := StringB;
'0123A56789'
```

Note : Since the target specification stipulates a single character replacement (not a slice) this is exactly what happens - as no index or slice is specified for StringB, the inserted character defaults to the first character.

```
StringA[4..4] := StringB;
'0123ABCDEFG56789'
```

Note : In this case, the target specifies a slice - although this slice consists of only one character, the replacement allows any number of characters to replace the nominated slice.

All of the above may also be combined with concatenation in the right hand side.

### Assignments to structures

Two extended assignment mechanisms are supported when assigning to structures.

The first is assignment of XML data into a structure. The second is assignment from Lava row types into an appropriately defined Row Type structure.

#### *Assignments from XML data into a structure*

Provided that the row type structure is correctly defined, XML data containing tags matching the structure elements will be assigned directly into the structure. For example, in the source code shown below, the assignment of the XML data as specified will result in the elements of the structure being assigned as follows :

```
Invoice_id : 2
Product_id : 23
LicenceQty : 999
Promotion_id : 0
Discount : 0.0
Amount : 999.0
bVatable : boolean FALSE
Licence_id : 0
```

Note that elements are matched with the tag name - the position (sequence) in the structure or the XML does not matter.

TYPE		=	ROWTYPE	
	DetailType			
	Invoice_id		:	LONGINT;
	Product_id		:	LONGINT;
	LicenceQty		:	LONGINT;

## Manual Scope and Target Audience

```

Promotion_id      : LONGINT;
Discount          : LONGREAL;
Amount           : LONGREAL;
bVatable         : BOOLEAN;
Licence_id       : LONGINT;

END;

PROCEDURE TestAssign;
VAR
  DetailStruc     : DetailType;
BEGIN
  DetailStruc := '<items> ' &
    '<formdata >      ' &
    '<Invoice_id>2</Invoice_id>      ' &
    '<Product_id>23</Product_id>      ' &
    '<LicenceQty>999</LicenceQty>      ' &
    '<Promotion_id>0</Promotion_id>      ' &
    '<Discount>          0.00</Discount>      ' &
    '<Amount>            999.00</Amount>      ' &
    '<bVatable>FALSE</bVatable>      ' &
    '<Licence_id>0</Licence_id>      ' &
    '</formdata>      ' &
    '</items>';
END TestAssign;

```

### *Assignment into structures from Lava object row types*

Lava object row types may be assigned into structures, provided that the structure is appropriately defined. In the following example, assuming that the Lava table has an identical definition to that of the row type, the assignment will occur on an element-by-element basis.

Table rows are assigned into structures in a strictly sequential manner - the first column in the table is assigned to the first element of the structure, and so on until the number of columns in the table has been exhausted or the structure runs out of elements.

```

TYPE
  DetailType = ROWTYPE
  Invoice_id      : LONGINT;
  Product_id     : LONGINT;
  LicenceQty     : LONGINT;
  Promotion_id   : LONGINT;
  Discount       : LONGREAL;
  Amount         : LONGREAL;
  bVatable       : BOOLEAN;
  Licence_id     : LONGINT;

END;

PROCEDURE TestAssign(pRow : INTEGER);
VAR
  DetailStruc     : DetailType;
BEGIN
  DetailStruc := LAVA.Accounts.InvoiceDetail[pRow];
END TestAssign;

```

**Assignments between structures**

Several forms of assignment between structure variables are supported.

*Element assignment*

Elements in structures may be assigned to one another. In this form, the normal rules of casting apply.

```
Struc1.element1 := Struc2.element3;
```

*Assignment between equivalent structures*

Given the following variable definition :

```
VAR
  DetailStruc1   :   DetailType;
  DetailStruc2   :   DetailType;
```

the following assignment may be performed :

```
DetailStruc1 := DetailStruc2;
```

In the case where the two structures are of the same type, the assignment is performed element-by-element with the first element of the source variable assigned to the first element of the target variable, and so on. Normal assignment rules apply to each element.

*Assignment between non-equivalent structures*

Where the variables are of different types, assignment may still be performed. Given the code segment below, the assignment of Var2 into Var1 will assign :

Var2.FormRow\_id to Var1.FormRow\_id  
 Var2.UnitCost to Var1.UnitCost  
 No assignment will be done to Var1.First.

The elements are assigned on a like-named basis, regardless of individual element type.

Normal casting rules apply to the assignments.

```
TYPE
  VariableData= ROWTYPE
    First      :   STRING[100];
    UnitCost   :   STRING[100];
    FormRow_id :   INTEGER;
  END;

  VariableSwitch = ROWTYPE
    FormRow_id :   INTEGER;
    UnitCost   :   STRING[100];
  END;

PROCEDURE Calc ();
VAR
  Var1      :   VariableData;
  Var2      :   VariableSwitch;
BEGIN
  Var1 := Var2;
END Calc;
```

### Conditions

Simple and compound conditions are supported according to standard boolean rules. Any comparison results in a boolean value, and thus boolean variables may be used in stead of a comparison, as in the following example

```
IF (Quantity > 500) OR bOverflow AND (Total = 33.5) THEN
END;
```

Note that in the equal comparison only a single equal sign is used, unlike the double equals (==) used in languages like C.

Block constructs in LavaStream are implicit; in other words the above IF statement automatically commences a block of statements without the requirement for a BEGIN as would be necessary in Pascal. For this reason, the END statement above is mandatory to close the block automatically commenced by the IF.

### Case statements

Complex case statements are supported, with two case constructors (range and alternative). The following example shows the basic variants.

```
CASE Index OF
| 33 :
| Limit :
| 42 .. 57 :
| 75, 87, 95 .. 100 :
ELSE
END;
```

Note that in contrast with languages like C, each case strictly contains the executed code. In other words, in the example above, if the value of Index is 33, the code contained between the commencement of case 33 and prior to case Limit will be executed, and the case terminates. No “break” clause is required - only one case clause at maximum can execute. If all specified cases do not apply, and an ELSE clause is provided, the code contained in the ELSE clause will execute.

As with the IF statement above, each case clause implicitly commences a block of code - no BEGIN..END is required.

### Loops

Four loop types are supported. These are :

- FOR loops      Loops with a specified counter, increment and limit
- WHILE loops    Loops with a preceding exit condition
- REPEAT loops   Loops with a succeeding exit condition
- Infinite loops    Unbounded loops

For all of the above, two control statements are supported :

CYCLE            skips the remaining statements in the loop and recommences at the start of the loop.

                  Increments and exit conditions still apply.

EXIT             exits the loop unconditionally.

In all of the above cases, the loop commencement statement automatically commences a block of code which is terminated by the end construct of the loop.

The following examples show basic syntax and application of loop types

```
FOR Index := 1 TO 10 DO
END;

FOR Index := Start TO End * 5 BY 3 DO
```

```
END;  
  
WHILE a = b DO  
END;  
  
REPEAT  
UNTIL a > b;  
  
LOOP  
  IF a > b THEN  
    EXIT;  
  END;  
END;  
END;
```

### Procedure calls

Procedure calls are as for Pascal or Oberon when the procedure is local to the module, as follows :

```
MyProcedure (Parameter1, Parameter2);
```

Similarly, function calls are an assignment into a variable :

```
LocalVar := MyFunction (Parameter1, Parameter2);
```

If the procedure or function is not local to the module, the call must be prefixed by the name of the imported module :

```
ImportedModule.ForeignProcedure (Parameter1, Parameter2);
```

As with assignment, it is permissible for the actual parameter (the parameters specified in the call to the procedure) to differ from the formal parameter type (the type of parameter in the definition for the procedure) as long as assignment between the types is permissible and supported.

### Return statement

The RETURN statement may be used both in procedures and in function-type procedures (procedures with a declared return type) at any point within the procedure to return to the calling procedure.

In the case of function-type procedures, the returned variable or constant need not be identical to the declared return type for the procedure, as long as casting from the returned type is supported by LavaStream :

```
RETURN 0;
```

will return to the calling procedure and return a value of 0 as the function value.

## Formal Procedure Declaration

The declaration of procedures in LavaStream is almost identical to that in Pascal or Oberon, with a few small differences.

```
PROCEDURE ExampleFunc (      pSession_id : INTEGER;  
                           VAR pObject_id  : INTEGER;  
                           pbProfile      : BOOLEAN;  
                           pSerialPort    : STRING  
                           ) : INTEGER;
```

## Manual Scope and Target Audience

Note that although the above procedure declaration is that of a function (the procedure returns an integer value) the prototype is still declared as PROCEDURE - there is no specific function type; functions are merely procedures which return values. However, this is not to be confused with the declaration of functions in languages such as Javascript where a “function” may return a value or not as the programmer sees fit - in LavaStream a function-type procedure must declare its intention to return a value in the prototype declaration as above, and such a function-type procedure must return a value when terminating, as in :

```
RETURN 0;
```

If a return without a specified return value is coded in a function-type procedure, this is a syntactical error.

Similarly, an untyped procedure (a non-function procedure) may not return a value, and any returns coded in the body of the procedure must be without any return value, as follows :

```
RETURN;
```

If, in a non-function procedure, a return value is specified this is also a syntax error.

Since the value specified in a return for a function-type procedure is identical to an assignment, the rules of automatic typecasting apply (see [Typecasting](#) for further details).

With the exception of the pObject\_id parameter in the example above, all the parameters will be passed by value. Parameters specified of form VAR are passed by reference.

Parameters which are specified by value form valid local variables in the procedure, and modification to these variables will not reflect on the calling procedure in any way.

Parameters specified as VAR (by reference) are assigned back to the referenced variable in the calling procedure whenever modification occurs. Although this is similar to the effect achieved when declaring a variable as a pointer reference in languages such as C (such as *int \*pObject* for example) the implementation is not exactly the same. LavaStream does not support pointers directly and no pointer arithmetic is supported by the language.

Unlike Oberon, it is permissible to specify a complex return type such as an array or structure, which provides more flexibility and makes it easier to abstract functionality.

### See also

[Omission of Pointers](#)

## Comments

Several forms of comments are supported. Both multi-line and single line comments are provided, and both types allow nesting to arbitrary depth

```
(* start of multi-line comment
  A := b;
// Single-line comment
End of multi-line comment *)
```

## External Function Calls

Although every effort has been made to provide as comprehensive a programming environment as possible in LavaStream, it is not possible to provide every possible function. For this reason, a mechanism is provided to allow external functions to be called from within LavaStream. Certain restrictions apply, but sufficient flexibility is provided that the majority of functions and parameters may be called easily.

## Manual Scope and Target Audience

In order to call an external function, this function must first be declared in LavaStream source. An example of such a declaration is provided below.

```
PROCEDURE [EXTERNAL:MyDLL] ExportFunc (
    pSession_id : INTEGER;
    VAR pObject_id : INTEGER;
        pbProfile : BOOLEAN;
        pSerialPort : STRINGPOINTER
    ) : INTEGER;
```

In the above example, the attribute `[EXTERNAL:MyDLL]` specifies that the procedure declaration is for an external function. The `MyDLL` portion identifies the DLL within which the function is to be found. This DLL must be available to the LavaStream runtime - typically this implies that the DLL must be present in the Windows System32 folder (commonly located at `C:\Windows\System32`). In addition, any DLLs required by `MyDLL` must also be present in this folder.

Both by-value and by-reference (VAR) parameters are supported for simple types, but note that STRING parameters are not supported - these cause complications in terms of the length of the string on the stack. Instead, the STRINGPOINTER parameter is provided - this will accept any conventional LavaStream STRING or a string constant, but will define a pointer to the string on the stack which must be accessed as such by the external function. By reference parameters are, in the case of external functions, defined as pointers to the nominated actual variable, and so the programmer must take appropriate care not to cause memory corruption in the external function when accessing these parameters. Note especially that specific storage details about LavaStream variables are intentionally not declared to permit modification to the design of the LavaStream engine without modification to the documentation - for this reason, accessing pointers in the form of by reference parameters in non-standard ways is not advised.

Simple return parameters are supported (integer, boolean, float) but not strings. Similar complications to those described above preclude this option. Pure procedures (no return value) are also supported.

Note that structures are not supported as parameters. LavaStream structures (in the form of ROWTYPE) are not size-specific, and are therefore not usable. Sized structures will be provided for this purpose in a future release of the LavaStream environment.

It is very important to ensure that the procedure declaration in the target function matches the external declaration in LavaStream exactly - if not, the system will undoubtedly corrupt the stack, and probably halt on some form of exception. It is not possible for the LavaStream compiler to validate the procedure declaration, and thus the onus of consistency checking is entirely the responsibility of the programmer.

The external function must be defined with the Pascal calling convention - in a language such as C, this implies a function of type `__declspec(dllexport) WINAPI` (or equivalent). Any appropriate language may be used, such as Borland Pascal or Visual C - note, however, that interface to class-based functions will not be possible. Also, as Java is not a machine language but an interpreted one, functions in Java cannot be called.

## Built-in Functions

This section provides a brief list of the built-in functions presented in the LavaStream language. For detail on the functions listed, see the LavaStream Functions chapter as well as the examples provided.

### Date and Time functions

<a href="#">Date</a>	Returns the current date in Julian format
<a href="#">DateFormat</a>	Specify the default date format
<a href="#">DateTime</a>	Returns the current date / time
<a href="#">Day</a>	Returns the day of the month for a given Julian date
<a href="#">Hour</a>	Returns the hour for a given Lava-format time
<a href="#">Minute</a>	Returns the minute of the hour for a given Lava-format time
<a href="#">Month</a>	Returns the month for a given Julian date
<a href="#">MsDate</a>	Returns the Julian date from a given Microsoft-format SQL timestamp
<a href="#">Second</a>	Returns the second of the minute from a given Lava-format time
<a href="#">Time</a>	Returns the current time
<a href="#">Year</a>	Returns the year from a given Juilian date

### Variable functions

<a href="#">ABS</a>	Returns the absolute value of a value or variable
<a href="#">BITAND</a>	Returns a bitwise and of two variables
<a href="#">BITOR</a>	Returns a bitwise or of two variables
<a href="#">BITXOR</a>	Returns a bitwise xor of two variables
<a href="#">CHAR</a>	Returns the ASCII character representing a given ordinal value
<a href="#">CLEAR</a>	Clears a variable to null values
<a href="#">DEC</a>	Decrements a variable by 1 or a specified amount
<a href="#">ENTIER</a>	Returns the integer part of a numeric variable
<a href="#">Even</a>	Returns true if the given number is even
<a href="#">INC</a>	Increments a variable by 1 or a specified amount
<a href="#">Length</a>	Returns the length of a given string
<a href="#">LOWER</a>	Returns a string in all lowercase

## Manual Scope and Target Audience

<a href="#">Odd</a>	Returns true if the given number is odd
<a href="#">ORD</a>	Returns the ordinal value of a character
<a href="#">ROUND</a>	Returns the rounded (integer) value of a variable
<a href="#">STRINGPOS</a>	Returns the position in a string of a string fragment
<a href="#">UPPER</a>	Returns a string in all uppercase

### Lava Database functions

<a href="#">AutoCommit</a>	Specifies autocommit for the current Lava session
<a href="#">Clear</a>	Deletes the nominated row of a given table
<a href="#">ColumnLabel</a>	Returns the label of the nominated Lava table column
<a href="#">COMMIT</a>	Commits any current open transaction frame
<a href="#">CREATETABLE</a>	Creates a lava table and returns the table id
<a href="#">DROPTABLE</a>	Drops a lava table
<a href="#">FindClose</a>	Closes a seek identifier
<a href="#">FindColumn</a>	Specifies a seek column for a seek identifier
<a href="#">FINDNEXT</a>	Find the next match for a column search
<a href="#">FindRow</a>	Find a row matching a given value for a nominated column
<a href="#">FindTable</a>	Specifies the table for a seek identifier
<a href="#">FREEROW</a>	Determine the first free row in a Lava table
<a href="#">LAVAERROR</a>	Return the description for a nominated error code
<a href="#">LAVASTATUS</a>	Return the status (success or error) for the last Lava or ODBC function call
<a href="#">OPENSESSION</a>	Open a new Lava session
<a href="#">RenameTable</a>	Renames the nominated Lava table
<a href="#">ReserveRows</a>	Reserves a specified number of rows for the nominated Lava table
<a href="#">ROLLBACK</a>	Perform a rollback on any open transaction frame
<a href="#">Session</a>	Returns the current Lava session
<a href="#">SETSESSION</a>	Set the current Lava session
<a href="#">SQL</a>	Perform an SQL command on the Lava Server
<a href="#">TABLECOLUMNS</a>	Return the number of columns in a nominated table
<a href="#">TableName</a>	Returns the name of the nominated Lava table

<a href="#">TABLEROWS</a>	Return the number of rows in a nominated table
---------------------------	--

### ODBC Database functions

<a href="#">ODBC_ADD</a>	Adds a row to an ODBC-linked database table
<a href="#">ODBC_CLOSE</a>	Closes the nominated ODBC connection
<a href="#">ODBC_CONNECT</a>	Connects to a nominated ODBC database
<a href="#">ODBC_SQL</a>	Perform an SQL command on an ODBC database
<a href="#">ODBC_UPDATE</a>	Performs an update to a nominated ODBC table row

### String Functions

<a href="#">Char</a>	Returns a character for a nominated ASCII value
<a href="#">IPfromString</a>	Returns an IP identifier (numeric) from a given string IP
<a href="#">Lower</a>	Converts a string to all lowercase
<a href="#">Ord</a>	Returns an ordinal value for a given character
<a href="#">StringFromIP</a>	Converts a numeric IP into a string
<a href="#">StringPos</a>	Finds the position for a substring in a string
<a href="#">TrimAll</a>	Trims spaces left and right of a given string
<a href="#">TrimLeft</a>	Trims spaces from the left of a given string
<a href="#">TrimRight</a>	Trims spaces from the right of a given string
<a href="#">Upper</a>	Converts a string to all uppercase

### Text output and formatting functions

<a href="#">FORMAT</a>	Returns a formatted output for numbers in currency, percentage, date or time format
<a href="#">OUTPUT</a>	Outputs text to the memory output buffer
<a href="#">TIME</a>	Returns the current time in millisecond after midnight format
<a href="#">XMLFOOTER</a>	Formats an XML segment footer
<a href="#">XMLHEADER</a>	Formats an XML segment header
<a href="#">XMLROWDATA</a>	Formats element data for a nominated rowtype

**Text file functions**

<a href="#">APPENDFILE</a>	Opens an existing text file for writing, and returns the handle
<a href="#">CLOSEFILE</a>	Closes an open file handle
<a href="#">CREATEFILE</a>	Creates a new file or opens and truncates an existing file, and returns the file handle
<a href="#">READFILE</a>	Reads text from an open file
<a href="#">SETFILEPOS</a>	Sets the byte position in an open file
<a href="#">SETPATH</a>	Sets the current default folder, and returns the previous default folder
<a href="#">SPACEFILE</a>	Spaces column output in an open file to a nominated column
<a href="#">WRITEFILE</a>	Writes text to an open file

**Windows related functions**

<a href="#">EXEC</a>	Executes a command, executable or batch file
<a href="#">MESSAGE</a>	Displays a message box
<a href="#">SLEEP</a>	Executes a windows sleep for a given period
WINFILEEXISTS	Determines whether a nominated file / path exists
WINFILECOPY	
WINFILEMOVE	
WINFILEDELETE	
WINFILERENAME	
WINGETLASTERROR	
WINFORMATMESSAGE	
WINAPPENDMENU	

## Program Structuring Mechanisms

LavaStream is designed to allow large programs to be designed and written. In order to support this, a layered approach to software design is supported. From the lowest level up, this provides procedures, modules and projects to be defined to allow a hierarchy of code to be structured.

The project layer is supported only in Blueprint, whereas the module and procedure layers are supported directly in the LavaStream system.

### Procedures

As with most high-level languages, LavaStream supports the definition of procedures with parameters in order to abstract certain functions or sets of instruction code. It is beyond the scope of this manual to define or explain the procedural mechanism in theoretical detail, and therefore the description below assumes a certain amount of programming knowledge and experience.

Procedures in LavaStream fall in two broad classes - private and public. Private procedures are visible only within a module, and cannot be invoked from external modules. Public procedures may be called from external modules, and allow extension of the programming paradigm for a given module not only to modules within the module's own project, but also to library procedures defined in foreign projects.

#### Procedure Declaration

A procedure is defined in terms of the declaration of a procedure framework, very similar to that in Pascal and identical to a definition in Oberon. An example follows.

```
PROCEDURE PureProcedure (parm1 : STRING; parm2 : INTEGER);
BEGIN
END PureProcedure;
```

The general procedure definition is provided by example below.

### Functions

Functions are defined exactly as procedures, with a specified return type. An example follows.

```
PROCEDURE FunctionProcedure (parm1 : STRING; parm2 : INTEGER) : STRING;
BEGIN
    RETURN parm1 [parm2..parm2+10];
END FunctionProcedure;
```

**See also**

[Formal Procedure Declaration](#)

### Modules

A module directly corresponds to a single text file, with the extension .lava, referenced from the Blueprint environment through a LavaStream Module icon. A module may contain exported and / or private declarations of constants, types, variables and procedures. Through use of multiple modules in a project, a project may be divided into more manageable segments of code which may cross-refer to one another through the IMPORT mechanism. Refer to the section on the Import Mechanism below for further details.

A module may be used to encapsulate a number of procedures which logically belong together, such as for example a set of procedures relating to one particular data table, or a particular functional area such as vector calculations. In large projects, dividing the definition of procedures in this way makes it easier to maintain the source code and also allows different programmers to work on different source code modules simultaneously without conflicting.

## Projects

LavaStream projects are made up of a set of LavaStream modules which should, at least nominally, be related to the functional responsibility of the project.

In the case of smaller projects, it may be that the project is entirely self-contained, and that the modules within the project fully define the functionality specified for the project.

For larger projects, the functionality may be divided between multiple projects, which use inter-project imports to access functionality defined in external projects in order to integrate the complete functionality required. See the section **Import Mechanism** below for further information.

A LavaStream project hierarchy is not limited in any way (except for mutual imports - see below) allowing extremely large software projects to be built through division into manageable projects each constructing part of the total functionality.

Due to the nature of the import mechanism, it is also possible to build libraries of functions which may be encapsulated within a project, for use by multiple other (perhaps unrelated) projects. There is no additional overhead at runtime when calling a function from an external project - such a function executes exactly as fast as a function in the local module.

## Import Mechanism

LavaStream implements an explicit import mechanism (similar to some Pascal systems, and almost identical to those used in Modula-II or Oberon) to allow access of entities (constants, variables, procedures) across module boundaries.

### Inter-module Imports

Unlike a language such as C, where external entities may be accessed freely provided some reference is present in (typically) an include file, in LavaStream only explicit references to external entities are provided. This means that all external references are prefixed by the name (or alias) of an explicitly imported module.

The following example illustrates in a simple way how the import mechanism works.

```
MODULE first;

CONST
(* Note in the following constant declaration the "-" before the "=", which
specifies an exported constant *)
  PI - = 3.14159265359;

END first.

MODULE second;

IMPORT first;
```

```
CONST
(* Note the use of the imported constant PI below, through the nomination
of the first module before the constant *)
  PIby2   = first.PI * 2;

END second;
```

In the above example you will see that the reference **first.PI** explicitly specifies the imported module as a prefix to the constant PI. It is not possible to refer to PI without this prefix - the main reason for this is the visibility provided by the mechanism. If a variable or procedure has no prefix, it is defined by the current module. If it has a prefix, it is defined in the module specified by the prefix. This makes source code more readable, and tracing the definition of an entity quite trivial.

Similarly to the above, cross-module references may be made to variables and procedures, allowing re-use of code and construction of very large projects with manageable division of source code.

Note that in any project where a module imports a second module, neither direct nor indirect mutual imports are supported. In other words, if module A imports module B, then module B may **not** import module A. Also, no module imported - directly or indirectly- by module B may import module A. Such mutual imports, direct or indirect, will result in a project which cannot be fully compiled.

### Inter-project Imports

In addition to importing modules within a given project, imports may be specified across project boundaries. This allows reusable functionality to be defined within a project which serves multiple other projects with this functionality.

For example, a function may be defined in Project ProjA (within Module ModA) which is to be used on Project B. In Module BB (within project B) the import would be specified as follows :

```
IMPORT
  ProjA.ModA;
```

where the **A**. nominates the project from which the module is to be imported, and the **AA** nominates the module to be imported from that project. Any entity visible within the project (as exported from a given module) is also visible from other projects.

Similarly to imports for modules within a project, cross-project imports may not cause any loops in the import chain. In other words, under no circumstances may a module from project A import a module from project B if that module (directly or indirectly) imports the same module from project A.

## Database Interfaces

LavaStream is primarily intended to provide application development for data-intensive requirements, typically those requiring at least one database. One of the aspects of the LavaStream environment which sets it apart from the majority of similar environments is the ability to interface - simultaneously and seamlessly - with multiple external ODBC databases as well as the integrated Lava database.

### Lava Server Interface

This chapter has only a superficial treatment of the facilities presented by the Lava database. For a more detailed specification of the operation and functionality in the Lava Server and the Lava Database (including Lava SQL) please consult the Qubik Lava Server Reference.

The LavaStream environment is tightly integrated with the Lava database. Array-like access (both read and write) to Lava data tables allows simple and powerful access to data, while a comprehensive SQL interface also allows complex statements and joins to be processed where a row-level interface does not provide adequate functionality.

The interface with the Lava database is different from a conventional application / database interface, in that the LavaStream environment does not operate as a conventional client in a client / server connection. Instead, LavaStream operates on a distributed client database, which allows solutions to be coded as if on an exclusively mounted single-user database. The distributed link to the Lava Server takes care of the rest. In addition, even if the link to the main server is across a slow WAN link, it is possible to add a Lava Satellite Server to the site configuration, allowing the LavaStream client to interface at LAN speeds to the local server, while a background communication process ensures that the Primary Lava Server is kept up-to-date.

In addition to the above, LavaStream has the ability to create temporary (virtual) tables on the distributed client database serving the LavaStream environment. This allows very high speed storage and retrieval of (potentially) huge amounts of transient data which may be required for processing purposes.

### SQL Interface

Where bulk operations are required, either to retrieve large result sets or to perform mass updates or deletions in the Lava database, a comprehensive SQL interface is provided which accepts any valid Lava SQL statement. A simple example is provided below.

```
ResultTable := SQL(
  'select ' &
    'customer, code ' &
  'from ' &
    'accounts.invoice ' &
  'where ' &
    'delivery_date = 0');
```

In the above example, the SELECT statement will return a Lava Table ID into the table variable ResultTable, which may be manipulated as for any Lava table - including row-level access as described below.

There are many occasions when interfacing with a database where SQL is the most convenient way of achieving a particular result. However, it is very often true that the primary requirement is to operate on a particular row in a table. When this is the case, SQL can be quite clumsy - compare the methods described below for providing row-level access to the database.

## Row-level Interface

In the majority of cases in a database application which presents a user interface to the database, access to the database is performed row by row. In these cases, performing the access via SQL is quite inconvenient. In order to present a more efficient interface, LavaStream allows direct integration with table data, as shown in the following simple example :

```
PROCEDURE FetchInvoice(pInvoice_id : INTEGER) : STRING;
VAR
  InvoiceRow      : LAVA.Accounts.Invoice.TYPE;
  ReturnXML      : STRING;
BEGIN
  (*.Retrieve the nominated invoice *)
  InvoiceRow := LAVA.Accounts.Invoice[pInvoice_id];
  IF InvoiceRow.DeliveryDate = 0 THEN
    InvoiceRow.Status := 'Undelivered';
  ELSE
    InvoiceRow.Status := 'Completed';
  END;
  (*.Update the database *)
  LAVA.Accounts.Invoice[pInvoice_id] := InvoiceRow;
  (*.Generate XML *)
  ReturnXML := XMLHEADER(TRUE, 'items');
  ReturnXML := ReturnXML & XMLROWDATA('invoicedata', InvoiceRow);
  ReturnXML := ReturnXML & XMLFOOTER('items');
  RETURN ReturnXML;
END FetchInvoice;
```

Note that in the above example little or no knowledge of the complete invoice row structure is required - in contrast with the requirement for a SQL statement. Thus, it is possible to write code which is relatively independent of the exact table design, thereby creating more reliable programs.

## Row-Level Seek

It is often necessary to locate a related row from given data, where nested processing in detail tables is to be performed (or for any other reason). In order to allow the location of row data directly without recourse to SQL, LavaStream presents a facility for finding rows in a data table directly, returning the row ID of the matching row(s).

Consider the following example :

```
PROCEDURE FindInv();
VAR
  Seek_id      : INTEGER;
  InvDtl      : LAVA.Account.InvoiceDetail.TYPE;
  Header      : LAVA.Account.InvoiceHeader.TYPE;
  Header_id   : INTEGER;
  Row_id      : INTEGER;
BEGIN
  FOR Header_id := 1 TO 100 DO
    Header := LAVA.Account.InvoiceHeader[Header_id];
    OUTPUT('Header : ' & Header.ID & ' ' & Header.Order_no & NEWLINE);
    Row_id := FINDROW( Seek_id,
                      LAVA.Account.InvoiceDetail,
                      LAVA.Account.InvoiceDetail.Order_no,
                      Header.Order_no);
    WHILE Row_id # 0 DO
```

```
    InvDtl := LAVA.Account.InvoiceDetail[Row_id];
    OUTPUT('      ' & InvDtl.ID & ' ' & InvDtl.Order_no & ' ' &
InvDtl.Stock_code & NEWLINE);
    Row_id := FINDNEXT(Seek_id);
  END;
  FINDCLOSE(Seek_id);
END;
END FindInv;
```

Note that the above example is not supposed to be very clever - the result would have been easier to reach through SQL. However, the functionality presented by the FIND / NEXT functions is clearly presented.

The above example illustrates a simple single-column seek. More complicated seeks based on multi-column conditions may also be executed - see the [FINDCOLUMN](#) command for an example of a multi-column seek and information on specifying inequalities as seek conditions.

### Lava Interface Selection

From the above information, it is clear that under some circumstances the conventional SQL technique will be more efficient. On these occasions, it is often best to simply use SQL. However, there are many occasions where SQL would be an inconvenient way to program a particular data access or update. In these cases, the row-level access, update and seek mechanisms may be used as an alternative to SQL in order to optimize the processing and achieve the result in a more direct manner.

Note that depending on the number of rows to be accessed, each of these techniques has advantages. Where a few rows (say 100 or less) are to be processed, it is highly likely that a row-level approach will be more efficient and will result in faster processing. For large numbers of rows to be retrieved or updated (almost certainly where thousands of rows are involved) SQL is likely to execute faster.

It is also possible to use both of these techniques together, provided that the row ID is selected when selecting the result set. In such a case, the row ID may be used to perform row-level operations or further exploration of related data, while the SQL data is used for the primary operations.

### Error checking

Directly after any Lava database function has been executed, the success or failure of the function may be checked using the LAVASTATUS function. The following code segment illustrates this technique :

```
rc := LAVASTATUS();
IF rc # 0 THEN
(*.Perform any error processing here *)
END;
```

The variable **rc** in the above example (an abbreviation for return code) will be 0 if the previous operation has succeeded, and non-zero if it failed for whatever reason. A textual explanation of the failure code may be obtained as follows :

```
ErrorText := LAVAERROR(rc);
```

## ODBC Interface

In order to allow for import from and export to foreign databases, LavaStream includes a comprehensive, high-speed ODBC interface.

## Manual Scope and Target Audience

In addition to the ability to execute SQL statements across the ODBC connection, LavaStream also has built-in facilities to allow simple insertion and update of rows in ODBC data tables.

To support large-scale import of data from an ODBC source, LavaStream provides for the creation of Lava data tables directly through the execution of an SQL statement through an ODBC connection. This allows rapid and flexible import of data from (potentially) multiple ODBC sources in order to construct consolidated databases from (potentially) multiple sites.

### Connecting to ODBC database servers

It is possible to connect to an ODBC server in one of two ways. In both cases, a Connect statement is issued; in one a so-called DSN (Data Source Name) is used, and in the other the driver's specific connect fields are entered directly. Compare the following examples :

```
Connection := ODBC_CONNECT('DSN=SQL_dsn');
```

Here, an existing DSN is used. Such a DSN must be created using the Data Sources (ODBC) dialog provided by Windows - this can normally be found in the Administrative Tools section of the Windows Control Panel.

```
Connection := ODBC_CONNECT('DRIVER=SQL Server;' &  
                            'SERVER=WindowsSQLserver;' &  
                            'UID=ValidUsername;PWD=CorrectPassword;' &  
                            'WSID=FredWorkstation;Network=DBMSSOCN');
```

In this case, the parameters for a Microsoft SQL Server are completed explicitly - this can normally be done for any ODBC Server, but the user must know the parameter names and required values in order to be able to construct the connect string correctly.

### SQL Interface

In almost all cases, the only interface an ODBC connection has to the ODBC Server is through SQL. LavaStream presents a number of pseudo-interfaces (as listed below) but these in fact all use SQL to interface to the ODBC database. Where these pseudo-interfaces do not provide the functionality required, a general-purpose SQL interface may be used, as shown in the following example :

```
Resulttable := ODBC_SQL(Connection,  
    "select " &  
        "invdtl.order_no, invdtl.order_line_no, invdtl.warehouse, " &  
        "invdtl.product, invdtl.weight " &  
    "from " &  
        "FAT.scheme.opdetm as invdtl");
```

The above example assumes that a valid ODBC connection has been obtained, and should use the correct SQL syntax for the target database.

A statement such as the one above will result in a Lava data table (the table ID for which is returned in the variable *Resulttable*). This result table may be processed as for any Lava table, with the single difference that the programmer must define the row type himself. For the above example, such a row type definition may look as follows :

```
InvoiceDetailType = ROWTYPE  
    Order_no      : STRING[50];  
    Line_no       : INTEGER;  
    Warehouse     : STRING[70];  
    Product       : STRING[100];  
    Weight        : FLOAT;
```

```
END;
```

It will also be necessary to declare a variable of the above row type, perhaps as follows :

```
VAR  
    InvoiceDetailRow      : InvoiceDetailType;
```

### Result Row Access

Having defined the row type as shown above, it is possible to retrieve individual result rows from the result table as follows :

```
FOR Index := 1 TO TABLEROWS(Resulttable) DO  
    InvoiceDetailRow := Resulttable[Index];  
    (*...Perform required processing here *)  
END;
```

Note that although nothing prohibits you from updating the result table (it is a table like any other) no such update will be reflected on the ODBC server - there is no link between the result table and its origin.

### Row Insertion

In order to provide a more reliable and less time-consuming method for inserting data into an ODBC data table, LavaStream presents the ODBC\_ADD function. This allows interaction with an ODBC data table based on the name of fields in a row type definition, with the LavaStream engine doing the majority of the work in terms of constructing the bulk of the SQL insert statement required.

Using the row type definition as above, a row may be added to the ODBC server as follows :

```
ODBC_Add(Connection, 'FAT.Scheme.opdetm', InvoiceDetailRow);
```

The implicit assumption is that the row type variable, InvoiceDetailRow, contains the values required to be added to the ODBC data table. Given that this is the case, the above instruction could generate SQL approximately as follows :

```
insert into FAT.scheme.opdetm  
    (order_no, line_no, warehouse, product, weight)  
values  
    ('01', 2, 'aa', 'prod1', 23.5)
```

### Row Updates

Similarly to the method above for inserting new rows, LavaStream also supports the update of existing rows in an ODBC table. In this case, the programmer has to specify an appropriate filter clause which will select the row to be updated.

Note that the filter specified must be adequate to select only the row to be updated - if the filter is incorrect or not selective enough, all rows selected will be updated. The onus is on the programmer to ensure that the filter will correctly select the appropriate row.

The example below illustrates the use of a row update, given the row types and variables as defined above :

```
ODBC_UPDATE( Connection,
```

```
'FAT.Scheme.invoice_lines',  
InvoiceDetailRow,  
'order_no = "01" and line_no = 2');
```

### Closing ODBC connections

Depending on the operations performed while connected to an ODBC source, a significant number of resources may be allocated to such a connection. In all cases it is a good idea to disconnect from the server when the connection is not in use - this allows LavaStream to free all allocated resources and also allows the ODBC Server to free any resources allocated to the connection. The following example illustrates how this is done.

```
ODBC_CLOSE(Connection);
```

If the connection is already closed, the function has no effect.

# Text File and Structure Interfaces

## Text File Interface

LavaStream provides a comprehensive text file interface, allowing the programmer to create or append to text files. In addition to the facilities provided for writing values, text files may also be read, while comprehensive string processing facilities allow formatting or decoding of string values.

### Writing Text Files

In order to write to a text file, the file can either be created (using the CREATEFILE function) or appended (using the APPENDFILE function). In both cases, if the file has been successfully opened, any string value may be written to the file using the WRITEFILE function.

If CREATEFILE is used to open an existing file, the file will be truncated (cleared) on open. If the content of the file should be preserved, use APPENDFILE instead. If an existing file is opened using APPENDFILE, the default file position will be at the end of the file, allowing text to be appended to the end of the file without further adjustment being necessary.

WRITEFILE allows any text to be output to the file - consistent with all other string processing in LavaStream, this means that any of the functions resulting in string output may be used, in conjunction with the automatic casting of values to text if appended / concatenated using the & operator.

Facilities such as FORMAT, STRINGFROMIP and UPPER / LOWER may be used in the parameter for WRITEFILE to create or format output for the text file.

In order to allow text files to be formatted in a columnar style, the SPACEFILE function allows the line output to be aligned on a nominated column to the right of current output.

### Reading Text Files

To read from a text file, the file should be opened using the APPENDFILE function - this will open an existing file without clearing the content.

In order to provide for reading input, the file pointer should first be set using the SETFILEPOS function - the default file pointer position for a file opened with APPENDFILE is at the end of the file.

Once the file has been opened and the file pointer set, successive READFILE functions may be called to read text data from the file. The READFILE function allows any desired termination string fragment to be specified, consisting of 1 or more characters on which the read operation is to terminate. This allows highly flexible input from text files.

It is possible and permissible to write text to a file which is being read - the onus is on the programmer to ensure that the data written will be correctly placed in the file. The file pointer is moved on as reads are performed, and always points to the next character to be read. If a WRITEFILE is performed at this point, it will overwrite the number of characters provided in the string specified, and will move on the file pointer by the same number of characters.

### Completing File I/O

Once all input or output has been concluded, the file should be closed using the CLOSEFILE function. This will ensure that all data buffered by the operating system has been written to the file, and will free the file for use by other programs.

## XML Interface

LavaStream provides a highly functional set of XML interface functions, providing for both input from and output to XML formatted data.

XML is rapidly becoming a very important data format - aside from long-standing use in the Internet environment as a method for communicating structured data, XML is now also the default format for Microsoft Excel.

### XML Import

In order to import XML data, it is necessary to have a structure correctly formatted and named to accept the data contained in the XML string.

For example, given the following XML data :

```
<?xml version="1.0" encoding="utf-8" ?>
<items>
  <formdata>
    <session>327</session>
    <name> John</name>
    <title>mr</title>
  </formdata>
</items>
```

It will be necessary to define a ROWTYPE structure corresponding to the XML data, similar to the example below :

```
FormRowType = ROWTYPE;
  Session   : STRING[100];
  Name      : STRING[100];
  Title     : STRING[20];
END;
```

Finally, a variable of the above type will be required, perhaps as follows :

```
VAR
  FormRow      : FormRowType;
```

Given the data and code segments as above, the following instruction will extract the XML data into the row type variable, assuming that the variable XMLstring contains the XML segment specified :

```
FormRow := XMLstring;
```

The LavaStream Runtime will automatically detect the XML content in the string variable nominated, and will parse the XML data into the row type variable.

Note that the exact sequence of the above row type does not have to correspond exactly to the XML data; the XML import functionality will match the individual data items based on naming rather than on position. Also, the match is not case sensitive, and will match the first field name which is equivalent regardless of case. It is important to note, therefore, that although LavaStream will permit distinct fields in a row type to be defined identically except for specific character case, an XML import with equivalent fields will successively match to the first field, leaving the second (case non-identical) field blank.

### XML Export

In order to output XML formatted text, a suitably defined ROWTYPE definition and variable are required - similarly to the requirement for XML import described above.

Given the same row type definition and variable declaration as in the above section, the following code segment could be used to output XML text :

```
Session := 327;
Name := "John";
Title := "mr";

(*.Outer tag *)
  OUTPUT(XMLHEADER(TRUE, 'items'));
(* Form group *)
  OUTPUT(XMLROWDATA('formdata', FormRow));
(*.End outer tag *)
  OUTPUT(XMLFOOTER('items'));
```

This code sequence will output an XML text segment identical to the XML segment specified above as the input for the XML import.

Note that conventionally the data for the row type variable would probably have been read from a data table, perhaps as follows :

```
FormRow := LAVA.SessionSchema.Session[Session_id];
```

where the table definition for the nominated table matches the field / column definitions in the row type.

# Programming Environment

The primary environment for developing LavaStream projects and source code is Qubik Blueprint. Blueprint is a graphical project environment, allowing extensive definition of both LavaStream projects as well as data dictionaries for use in LavaStream projects and Web-based form definition for auto-generated application development.

This chapter is a brief introduction to the Qubik Blueprint environment. For a more detailed coverage of the methods and techniques used when programming using Qubik Blueprint and LavaStream, see the Qubik Blueprint Operations Guide.

## Blueprint Operating Principles

Unlike the majority of other software development environments, Blueprint is based entirely on a central database. All projects, attributes and even source code are stored centrally in a Lava database, and all developers working in Blueprint access the same data from the same database. Although it is possible to configure Blueprint to place a copy of the source code on your workstation, the reference copy of the source is kept in the database and is accessed and updated by all developers logged in to the database. All programmers see the latest version of the source code, and operate on the latest project definitions and data dictionaries.

As LavaStream is an interpreted language which runs from a database image, compiled source (a LavaStream module image) is also stored in the central database.

There are, then, no files stored on your workstation unless you specifically configure Blueprint to place copies in a nominated folder. Everything comes from the central Lava Server.

## Programming using Qubik Blueprint

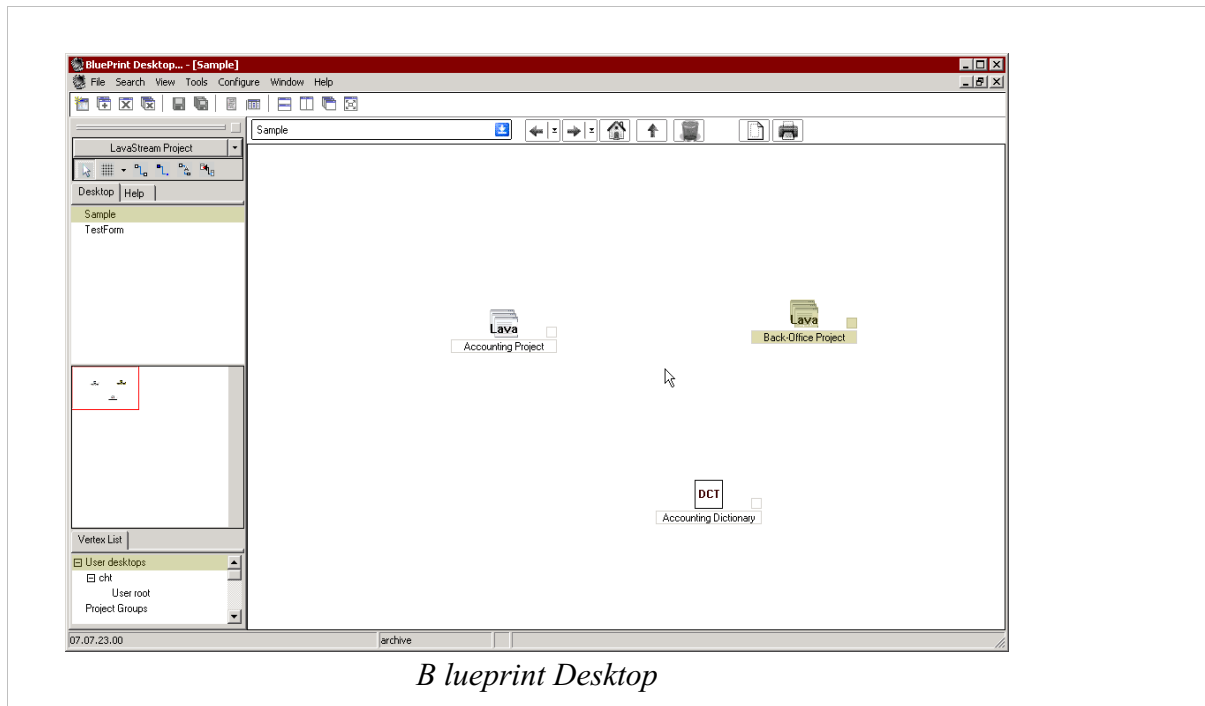
The Blueprint Development Environment allows creation and management of both LavaStream Projects and Lava Data Dictionaries, ranging from small to extremely large.

### Graphical Object Management

The Blueprint Development Environment is a graphical environment, which allows definition and management of projects and modules through a graphical depiction of these objects through the use of nodes on a programming desktop.

The environment as viewed at any one level allows definition (creation), viewing and setting of attributes for a number of objects or nodes which may include LavaStream Projects, Dictionaries, or - below project level - LavaStream Modules.

An example of a desktop layout including several defined objects is depicted below.



*B lueprint Desktop*

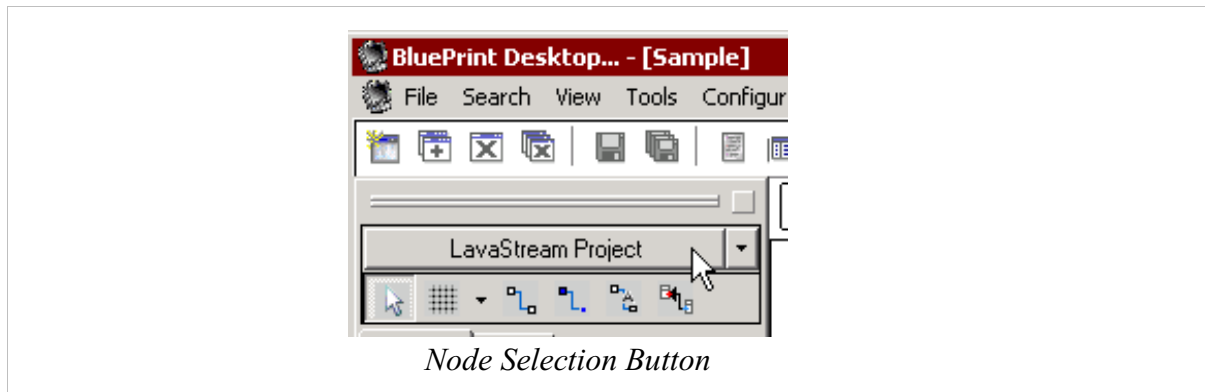
The above example depicts two LavaStream Projects (Accounting and Back-Office) and a single dictionary (Accounting).

The desktop depicts a layer in the design, and it is possible to descend through a node to the layer below. In the above example, double-clicking on a node or clicking on the node hotspot (a small rectangle to the bottom right of the node, just above the node label) descends to the layer below. In the case of a LavaStream project, this layer will probably contain one or more LavaStream modules, or perhaps further projects in the case of a larger project hierarchy. In the case of a dictionary, the lower layer will probably contain several table nodes, and perhaps also one or more subordinate dictionary nodes.

The total desktop is much larger than the viewing area, and may be viewed by using the desktop map presented to the left of the desktop. The small red square in the map represents the viewing area. The viewing area may be moved by clicking and dragging the red square in the map and sliding it to any desired area on the total desktop. The viewing area will follow the map to the desired region.

### Creating Nodes

In order to create a new node on the current Blueprint desktop, simply click on the node selection button as depicted below :



A drop list will appear listing the available node types. Select the appropriate node type, and click on the desktop to create the node.

Once the node is created, it should be re-named to something appropriate. To do this, select the node by clicking on it (the node will highlight) then press F2 to edit the label. Press **Enter** when done.

### LavaStream Projects

In order to group related functionality and to allow management of large-scale software development, LavaStream modules are grouped into projects. Through this mechanism, combined with the ability to define hierarchies of projects (projects containing lower-level projects, perhaps through multiple levels) it is possible to divide very large projects into more manageable sub-projects which may be used as parts or libraries to construct the full set of required functionality.

LavaStream has the ability to import functionality both from other modules within a project, as well as from modules belonging to other projects. The combination of the provision for projects at multiple levels and the ability to import functionality from subordinate projects means that software projects of almost arbitrary complexity may be constructed.

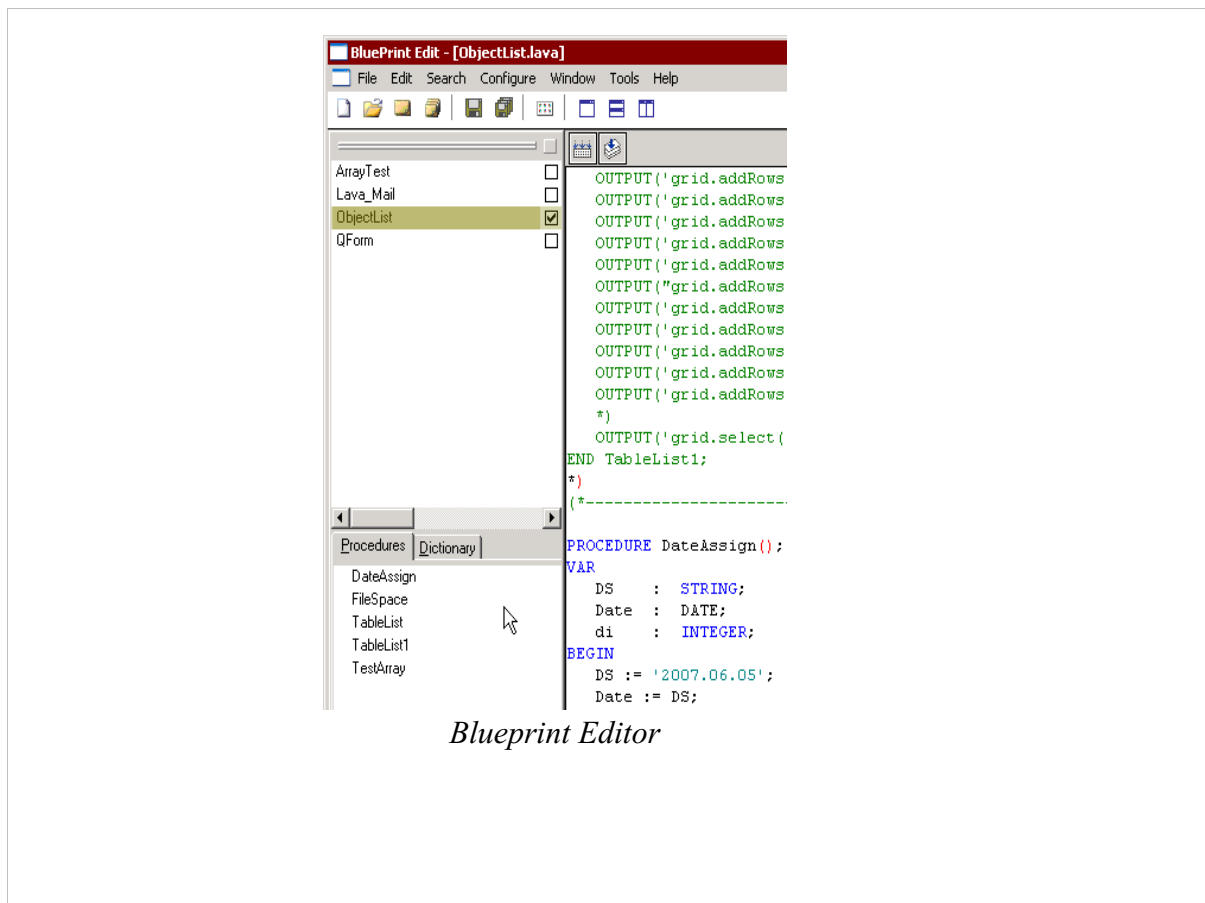
### LavaStream Modules

A LavaStream module is represented by a module node on the Blueprint desktop, and contains software source code in the form of LavaStream programming. This source code is stored in the main Lava database, and is available to all Blueprint users regardless of which workstation they are operating on.

### Editing and Compiling LavaStream Modules

In order to edit a LavaStream module, right-click on the LavaStream module node and select **Edit Source** from the pop-up menu. This will invoke the Blueprint Editor, which allows modification and compilation of source code.

Editing in the Blueprint Editor is very similar to other multi-source editors. There are a few facilities which provide convenient functionality worth knowing about.



In the partial image of the Blueprint Editor above, the following facilities are important :

#### Module List

The list in the top left corner of the image is the module list. These are the source modules currently loaded in the editor, and any particular module may be selected by clicking on it with the mouse. It is also possible to select a module by first switching to the module list by pressing **Ctrl-F** (for File), then selecting the desired module either using the up or down arrow keys or by pressing the first letter of the name of the module, then pressing **Enter** when the desired module is highlighted.

#### Procedure List

The list on the lower left of the image is the procedure list for the current module. These are all procedures defined in the source module, and you may navigate to any one of these procedures by clicking on the procedure name with the mouse. It is also possible to select a module by first switching to the module list by pressing **Ctrl-P** (for Procedure), then selecting the desired procedure either using the up or down arrow keys or by pressing the first letter of the name of the procedure, then pressing **Enter** when the desired procedure is highlighted.

#### Compile Button

Above the source code and just to the right of the module list is the compile button (the leftmost of the two buttons presented). The current module may be compiled by clicking on this button.

### Dictionary Definition

The Blueprint Dictionary system allows definition of named dictionaries, which serve as an encapsulating

## Manual Scope and Target Audience

mechanism for defining one through many data tables which are part of the dictionary.

In addition, the dictionary system allows hierarchies of dictionaries. If the intended dictionary will be too large to be defined in a single dictionary page, Blueprint allows subordinate dictionaries to be defined within a dictionary, and will generate all data tables at all levels within a dictionary hierarchy as if they belong to the topmost dictionary in the hierarchy.

In addition to allowing for very large dictionaries, this mechanism allows dictionaries to span any number of schemas in the database. Each dictionary nominates the schema into which the tables defined will be placed, but subordinate dictionaries do not have to nominate the same schema, and may place their tables in any required schema.

Within any dictionary in a hierarchy of dictionaries, an alias (synonym) table may be defined for any other table in any other dictionary, to allow for relations to be defined between tables in different dictionaries. See **Definition of Relations** for further details.

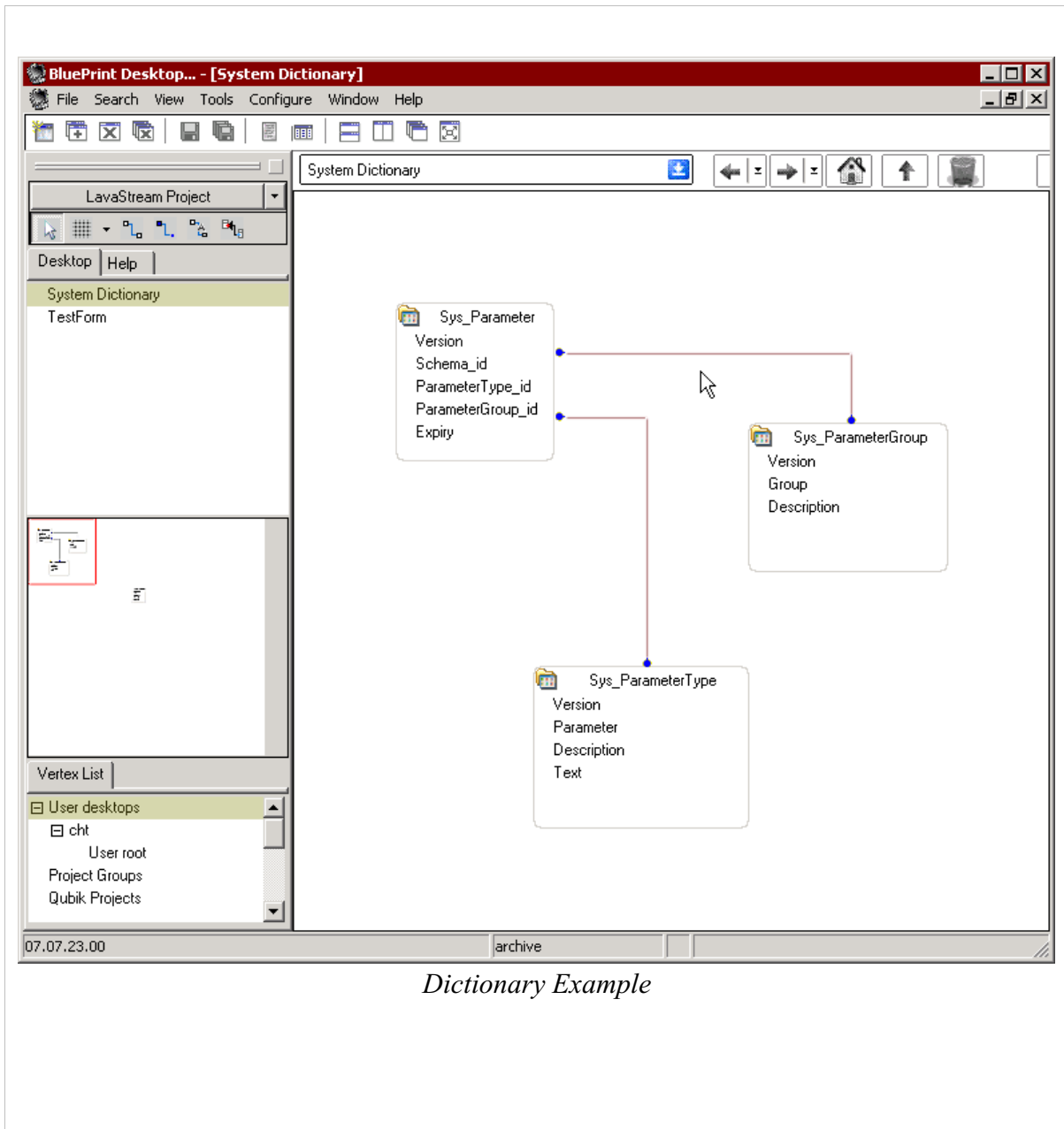
### Table Definition

Within any dictionary, data tables may be defined. A table may be defined from the ground up, by creating a dictionary table icon through the toolbar, after which attributes and columns for the table may be defined through the table property interface.

Alternatively, if the table currently exists in the Lava Database (perhaps due to restoring a backup, or due to creation of the table through SQL commands) such a table may be imported into the dictionary by using the table import mechanism accessed by right-clicking on the Blueprint desktop and selecting the **Import** option.

A table may be created on the desktop by clicking on the node creation button to the top left of the Blueprint desktop, then selecting **Dictionary Table** from the list. Note that tables may only be created below a dictionary node - dictionaries are the encapsulating entity for tables, and allow the generation of SQL for the creation of the tables contained in the dictionary.

The following image depicts a small dictionary as defined in Blueprint.



*Dictionary Example*

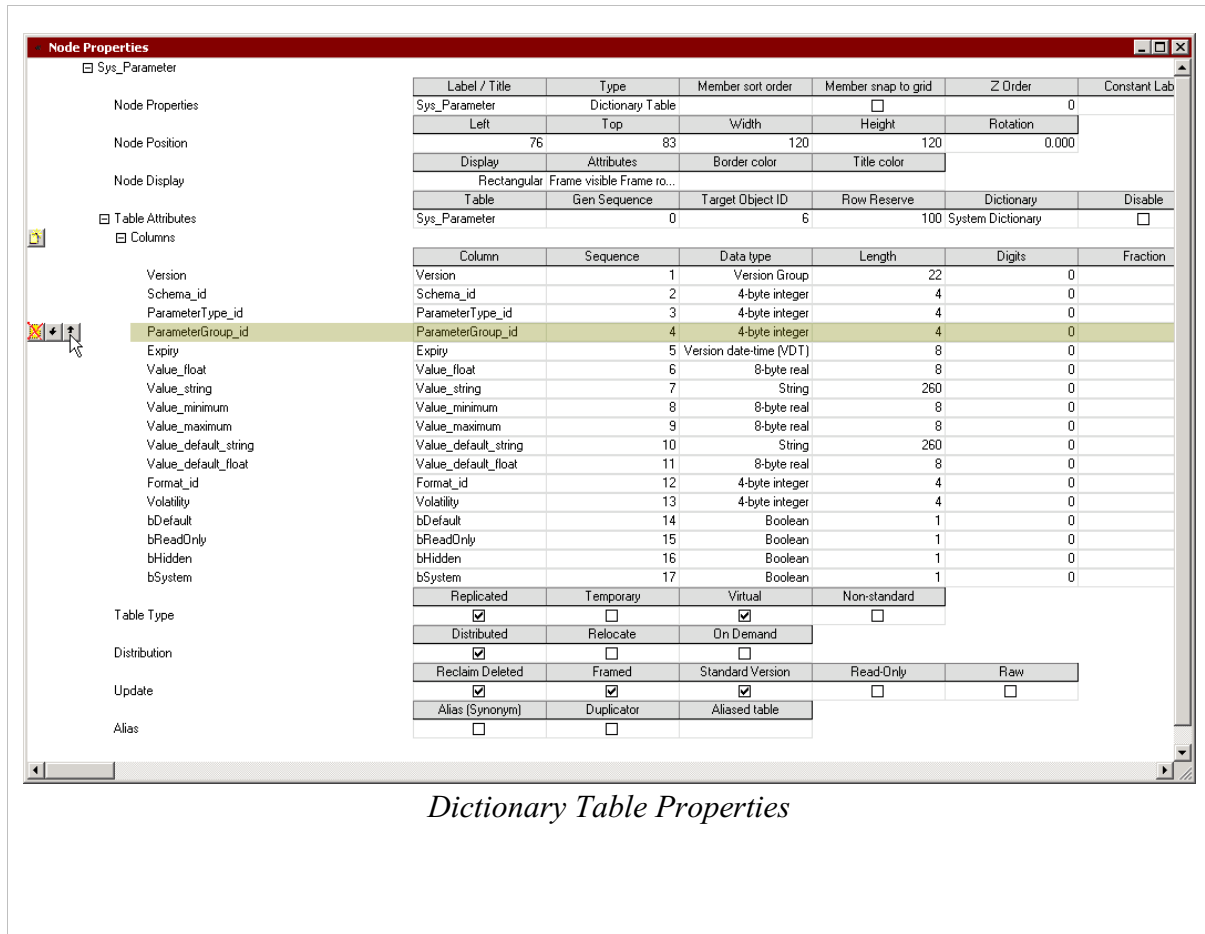
In the above example, three tables are depicted in a dictionary called “System Dictionary”, and two relations (the lines between the tables) are shown. There is also a fourth table in the off-viewing area, as can be seen in the map.

### Column Definition

Columns may be defined or amended for any dictionary table by right-clicking on the table node and selecting Table Properties from the pop-up menu.

The Table Properties window allows comprehensive definition of the table. An example is presented below,

using the Sys\_Parameter table from the above dictionary image.



*Dictionary Table Properties*

In the above image, the attributes for the table may be seen toward the bottom of the image (Table Type, Distribute, Update, Alias) - these properties are described in detail in the Qubik Blueprint Operation Guide.

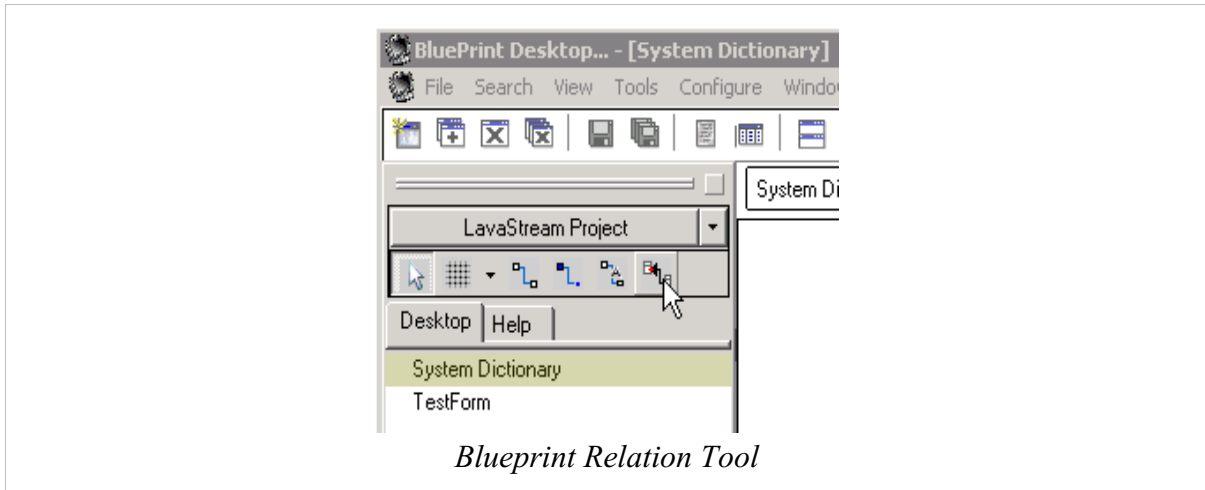
The columns for the table may be seen directly above the attributes. The column label, sequence number and type may be seen - there are a number of other column attributes toward the right of the window (not shown).

At the mouse position are three buttons; from left to right : Delete column, Move down, Move up.

A little higher up,, to the left of the **Column** tree label, is the Column Create button. This allows addition of new columns into the column list.

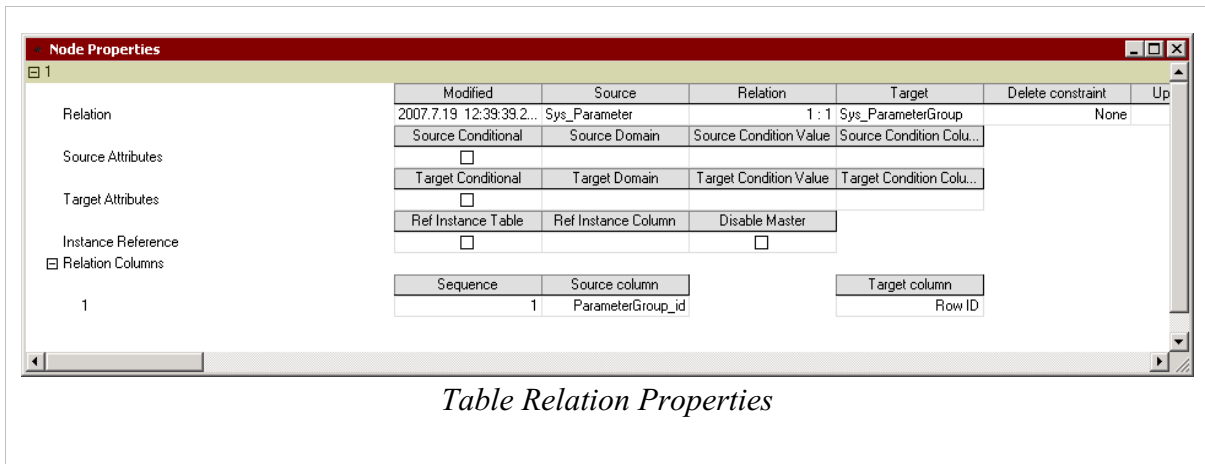
## Definition of Relations

To define a new relation between two tables, first select the Relation tool from the tool list, shown under the mouse in the following example :



Once the relation tool has been selected (the mouse cursor will change) a relation may be defined by clicking on the first table node, then dragging to the second table node and releasing the mouse over it. A relation line will be drawn between the two nodes.

The relation may be fully defined by right-clicking on the relation line and selecting **Relation Properties** from the pop-up menu.



Using the Relation Property window, the type of relation and the columns representing the relation may be defined. Advanced relation properties are discussed in the Qubik Blueprint Operation Guide.

## Generating a Dictionary

## Creating Data Tables on a Lava Server

## Integrating LavaStream with a Dictionary Schema

## Testing using LavaStream Elements

The LavaStream Elements test environment allows execution of LavaStream procedures directly without recourse to the Lava Scheduler or invocation through Apache Web Server. This environment is used primarily for two purposes :

- Evaluating LavaStream procedures in a standalone test environment
- Executing batch procedures written in LavaStream which perform data imports or data modification

In the first case, a LavaStream procedure may be executed in order to examine the results of the execution either in text form in the output editor, or to check data output using the database object viewer.

In the second case, LavaStream Elements is merely used as a means of invoking a procedure, optionally with parameters, in order to execute the procedure in a similar way to clicking on an icon on the Windows desktop or invoking a program from the start menu.

In an alternative to the above invocation, it is possible to configure an icon in the Blueprint Desktop which will invoke the procedure optionally with nominated parameters.

### Adjusting the Elements Mount Parameters

LavaStream Elements may be mounted in one of two ways. The first (and default) is client mode. This yields the same form of connection presented within the Blueprint Editor for executing LavaStream procedures. The second is exclusive mount. This allows fast execution of procedure without the overhead of distributed data transmission between the client and the Lava Server.

#### Running in Exclusive Mount

To use LavaStream Elements in exclusive mount, the parameters to the Elements executable need to be modified. You may want to create a second copy of the shortcut in order to retain the default (client) mount parameters.

The default parameters may be found by first expanding the Start Menu to the LavaStream Elements entry below the Qubik menu, then right-clicking on the entry and selecting the “Properties” option. The parameters are presented in the “Target” entry field of the Properties dialog.

If you wish to make a copy of the Start Menu entry for the Elements application, you may click-and-drag the Start Menu entry to the desktop, then hold Shift+Control while dropping the entry on the desktop. This will form a new shortcut to the application.

Once you have made a copy of the shortcut, the parameters may be modified from the default by right-clicking on the shortcut and selecting the “Properties” option. Modify the “Target” entry from the default, something like this :

```
/ClientPath=S:\Temp\lava /User=Fred /Password=BloggsPass /server=LavaServer
```

After modifying the above parameters to exclusive mount mode, the parameters could look as follows :

```
/User=Fred /Password=BloggsPass /Datapath=D:\Qubik\LavaPrimary
```

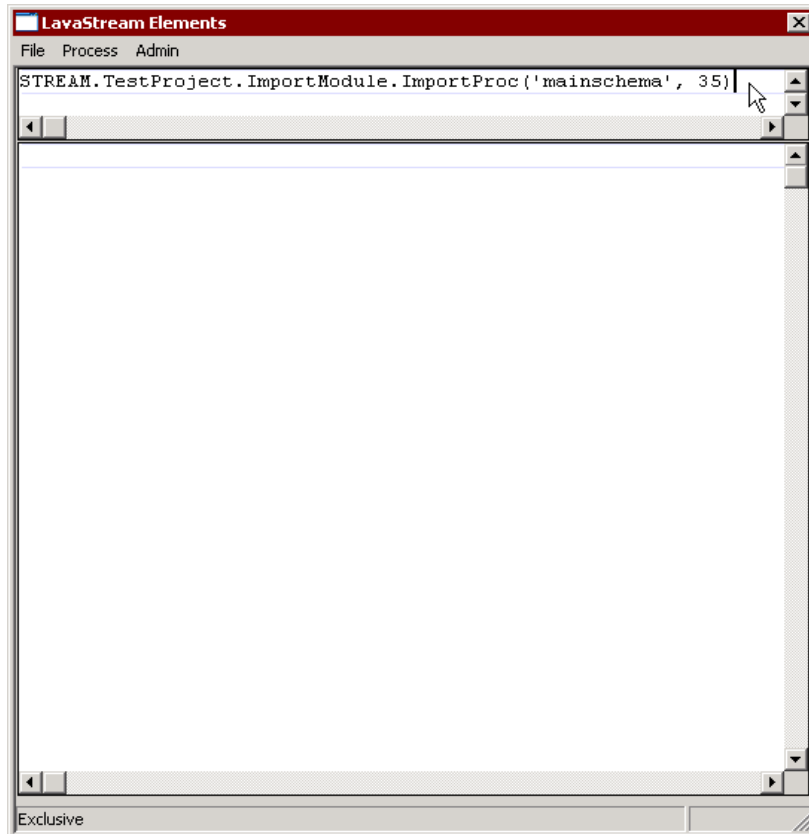
(Assuming that the main database is located at the path *D:\Qubik\LavaPrimary*).

Should it be appropriate, you may also wish to change the login parameters to the system account (the default would be user system, password manager - the password may have been changed by your database

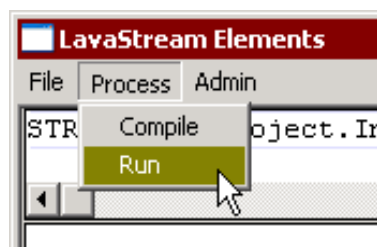
administrator).

## Executing a LavaStream Procedure

Once you have connected to the database with LavaStream Elements (either in client mode or in exclusive mode, depending on requirements) the procedure to be executed may either be typed into the command section of the main interface (the upper edit box in the window depicted below) :



In the above example, the procedure `ImportProc` which accepts two parameters, a string and an integer, is invoked as follows :

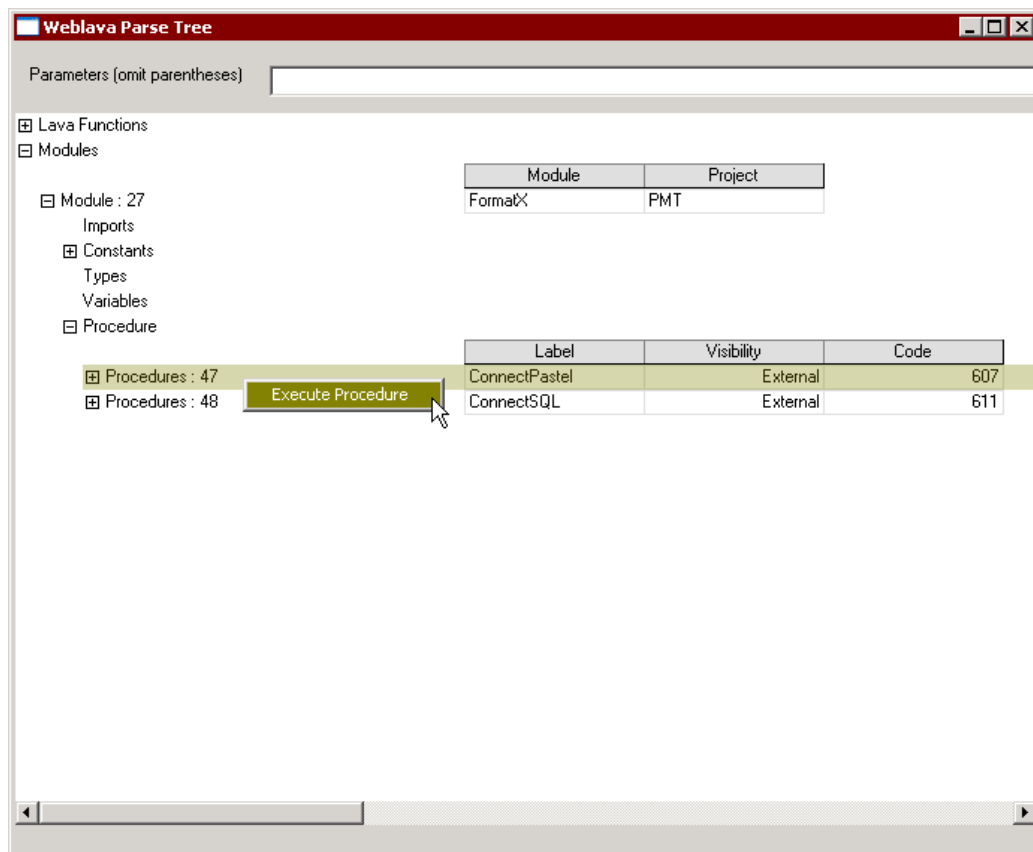


Note that the specification of the procedure to be invoked is case sensitive, and must therefore match the definition of the project, the module and the procedure exactly.

Provided that the command is entered correctly, the procedure will be executed and any output produced by the procedure will be displayed in the lower edit box.

### Selecting a LavaStream Procedure for execution

Instead of typing in the command to execute the procedure, it is possible to select the procedure from a procedure tree. The tree interface may be invoked by selecting *Admin / Parse Tree* from the LavaStream Elements menu. Once the parse tree has been expanded to the required module, a procedure may be invoked, as follows :



The *Execute Procedure* option may be selected by right-clicking on the entry for the required procedure.

Should any parameters be required by the procedure, these may be typed into the entry field at the top of the window.

### Viewing Output

After completion of execution, any output generated using the OUTPUT command in LavaStream will be reflected in the lower of the two edit boxes in the main LavaStream Elements interface window. The edit box is not limited in any practical way, and will permit large output buffers to be displayed in a scrollable window.

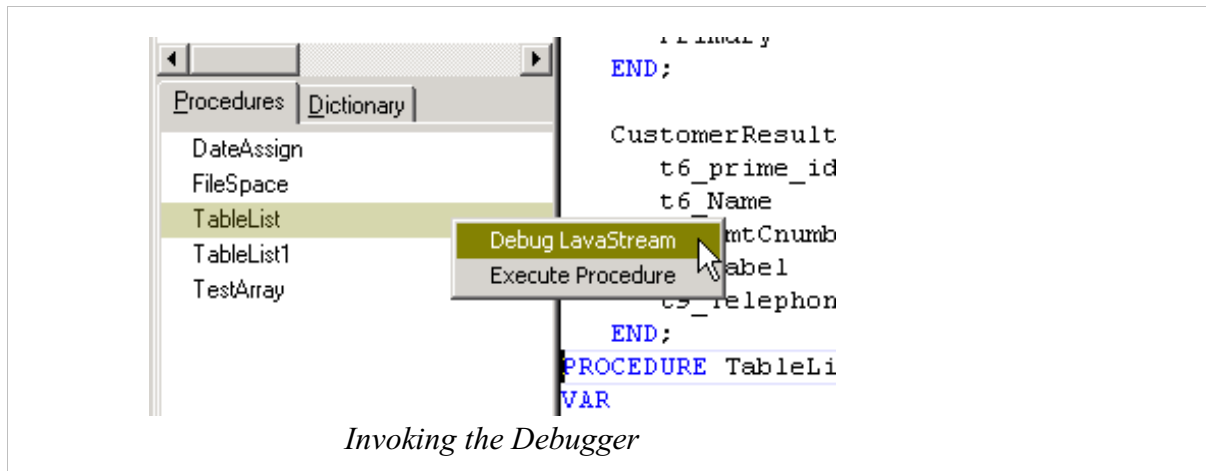
## Debugging LavaStream

The LavaStream debugger is integrated into the Blueprint Editor. To invoke the debugger on a procedure, first

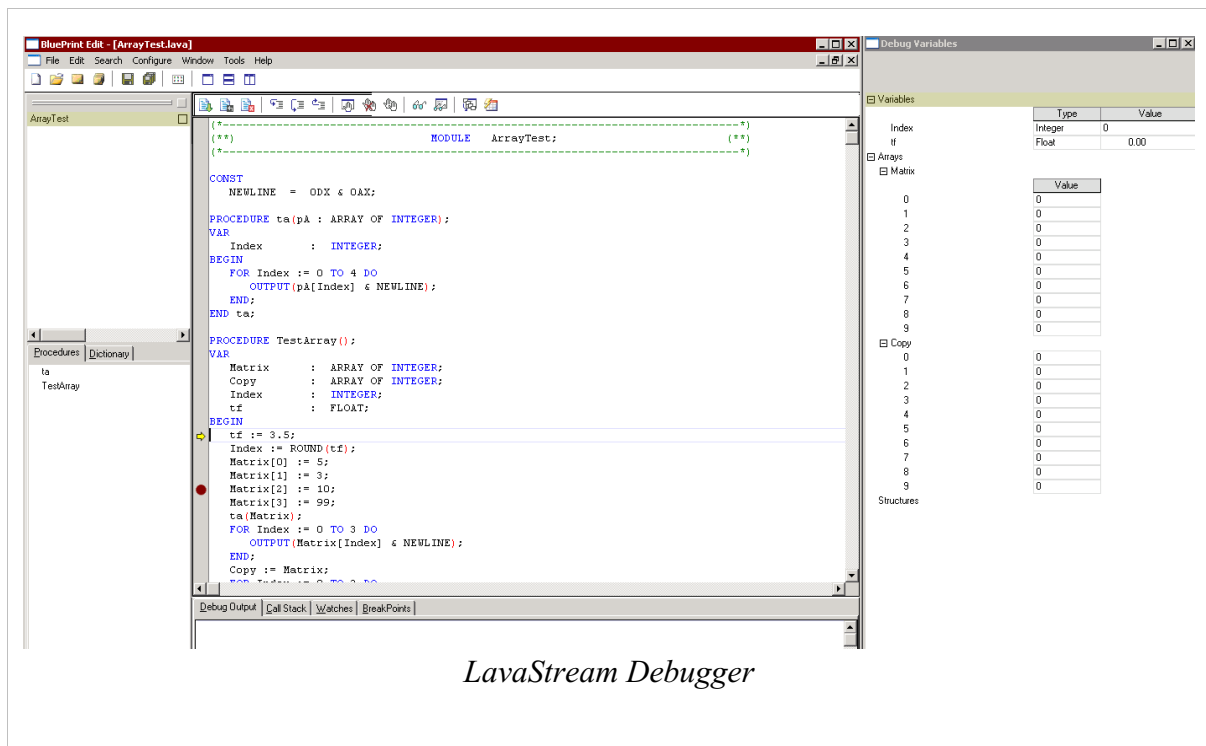
## Manual Scope and Target Audience

edit the module in which the procedure is defined. Be sure to compile the module before invoking the debugger, and check that the module compiles without errors.

To invoke the debugger, right-click on the procedure in the procedure list, and select the **debug lavastream** option



Provided the procedure is consistently compiled and without errors, the debugger will start at the first instruction of the selected procedure. A typical depiction of the debugger interface is shown below :



Once the debugger has been invoked, the following facilities are available :

### Examine Variables

The Debug Variables window will display all variables - parameters, local variables, array variables and module

variables - which are in scope. These are divided into three sections :

### Variables

Displays all simple variables in scope.

### Arrays

Lists array variables. The currently defined elements of the array are shown - as array size is dynamic, this will be the size of the array as defined by the highest index of the array used to this point in the code.

### Structures

Lists structure variables - both row type variables and Lava table variables. The full content of the structure is displayed.

## Setting Breakpoints

Breakpoints may be set at any point in the executable source code. To set a breakpoint, either click in the gutter at the line where the breakpoint is required, or move the cursor in the source window to the line on which the breakpoint is to be set and press F9.

Once a breakpoint has been set, parameters applying to the breakpoint may be adjusted in the breakpoint tab below the source code window. An example is shown below :

Debugger Breakpoints						
	Module	Procedure	Condition	Count	Disable	Encountered
1	ArrayTest	TestArray		0	<input checked="" type="checkbox"/>	0
2	ArrayTest	ta	Count =	1	<input type="checkbox"/>	0
3	ArrayTest	TestArray	Count multiple of	2	<input type="checkbox"/>	0

*Debugger Breakpoints*

From the example it can be seen that breakpoints may be :

### Unconditional

An unconditional breakpoint triggers whenever the execution pointer reaches the nominated source line

### Disabled

A disabled breakpoint does not trigger, and is functionally equivalent to not having a breakpoint. If the breakpoint is re-enabled, it is once again active.

### Conditional

Breakpoints may be set as conditional on the basis of the number of times the breakpoint is triggered. The breakpoint only triggers when the condition becomes true.

## Executing the Code

Code execution may be achieved either through the toolbar buttons at the top of the debugger source window, or through the function key shortcuts as listed below.

Note that variable display is only refreshed when the debugger is halted.

### **Step Over - F10**

This will step to the next source line in sequence, stepping over any procedure calls encountered.

### **Step Into - F11**

If the current source line is a procedure call, the debugger will step into the called procedure.

### **Step Out**

The debugger will run to the end of the current procedure, and break on the first line of code following the call to the procedure.

### **Go to Breakpoint - F5**

The code is executed until the first valid breakpoint is encountered.

The specific function keys were selected as these function keys (with some variation) have become a de facto standard in debuggers. The most common assignments were selected.

## **Viewing Output**

While debugging code, and at every break in debug execution, the debugger updates the output window as depicted below.

Output is generated through the LavaStream OUTPUT command, as in the following example :

```
OUTPUT('Matrix element ' & Index & ' : ' & Matrix[Index]);
```

Using this technique additional information on execution may be obtained than is visible from the variable window. The Output command as above is also used to generate output when in the Apache environment; i.e. all HTML and XML output to the browser client is generated in this way.

```

Matrix[2] := 10;
Matrix[3] := 99;
ta(Matrix);
FOR Index := 0 TO 3 DO
    OUTPUT('Matrix element ' & Index & ' : ' & Matrix[Index] & NEWLINE);
END;
Copy := Matrix;
FOR Index := 0 TO 3 DO
    OUTPUT(Copy[Index] & NEWLINE);
END;
END TestArray;

(*-----*)
END ArrayTest.

```

Debug Output | Call Stack | Watches | BreakPoints

```

t element 0 : 5
t element 1 : 3
t element 2 : 1
t element 3 : 99
t element 4 : 0
Matrix element 0 : 5
Matrix element 1 : 3
Matrix element 2 : 10
Matrix element 3 : 99

```

*Debugging Output*

### Exit the Debugger

The debugger may be exited either by clicking on the button in the toolbar or by pressing Ctrl-B. This will terminate the debug session and restore the editor to its usual state. The debugger may be re-invoked, or other editing functions may be performed.

# LavaStream Syntax

## Formal Syntax

The following formal syntax definition is very similar to that for the programming language Oberon, and also has large similarities with Pascal and Modula. In general, the language is somewhat simplified as compared with Oberon, but obeys the same design principles.

ident	=	letter {letter   digit}.
number	=	integer   real.
integer	=	digit {digit}   digit {hexDigit} "H" .
real	=	digit {digit} "." {digit} [ScaleFactor].
ScaleFactor	=	("E"   "D") ["+"   "-"] digit {digit}.
hexDigit	=	digit   "A"   "B"   "C"   "D"   "E"   "F".
digit	=	"0"   "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9".
CharConstant	=	' ' character ' '   digit {hexDigit} "X".
string	=	' ' {character} ' ' .
qualident	=	[ident "."] ident.
identdef	=	ident ["*"].
TypeDeclaration	=	identdef "=" RowType.
RowType	=	ROWTYPE FieldListSequence END.
FieldListSequence	=	FieldList {";" FieldList}.
FieldList	=	[IdentList ":" type].
IdentList	=	identdef {"," identdef}.
VariableDeclaration	=	IdentList ":" type.
designator	=	qualident {"." ident   "[" ExpList "]"   "(" qualident ")"   "^" }.
ExpList	=	expression {"," expression}.
expression	=	SimpleExpression [relation SimpleExpression].
relation	=	"="   "#"   "<"   "<="   ">"   ">="
SimpleExpression	=	["+   "-"] term {AddOperator term}.
AddOperator	=	"+"   "-"   OR .
term	=	factor {MulOperator factor}.
MulOperator	=	"*"   "/"   DIV   MOD   "&" .
factor	=	number   CharConstant   string   NIL   set   designator [ActualParameters]   "(" expression ")"   "~" factor.
ActualParameters	=	"(" [ExpList] ")" .
statement	=	[assignment   ProcedureCall   IfStatement   CaseStatement   WhileStatement   RepeatStatement   LoopStatement   [expression] ].
assignment	=	designator "!=" expression.
ProcedureCall	=	designator [ActualParameters].
IfStatement	=	IF expression THEN StatementSequence {ELSIF expression THEN StatementSequence} [ELSE StatementSequence] END.
CaseStatement	=	CASE expression OF {" " } case {" " case} [ELSE StatementSequence] END.
Case	=	[CaseLabelList ":" StatementSequence].
CaseLabelList	=	CaseLabels {"," CaseLabels}.
CaseLabels	=	ConstExpression [".." ConstExpression].
WhileStatement	=	WHILE expression DO StatementSequence   LoopControl END.

## Manual Scope and Target Audience

LoopStatement	=	LOOP StatementSequence   LoopControl END.
RepeatStatement	=	REPEAT StatementSequence   LoopControl UNTIL expression ";"
ForStatement	=	FOR ident ":=" Expression TO Expression [BY ConstExpression] DO StatementSequence   LoopControl END.
LoopControl	=	CYCLE   EXIT
ProcedureDeclaration	=	ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading	=	PROCEDURE identdef [FormalParameters].
ProcedureBody	=	VAR {VariableDeclaration ";" } [BEGIN StatementSequence] END.
ForwardDeclaration	=	PROCEDURE "^" identdef [FormalParameters].
FormalParameters	=	"(" [FPSection {";" FPSection}] ")" [":" qualident].
FPSection	=	[VAR] ident {"," ident} ":" FormalType.
FormalType	=	qualident.
DeclarationSequence	=	{CONST {ConstantDeclaration ";" }   TYPE {TypeDeclaration ";" }   VAR {VariableDeclaration ";" } {ProcedureDeclaration ";"   ForwardDeclaration ";" }.
Module	=	MODULE ident ";" [ImportList] DeclarationSequence [BEGIN StatementSequence] END ident "." .
ImportList	=	IMPORT import {"," import} ";" .
Import	=	ident [":=" ident].

# LavaStream Functions

A wide range of functions are natively supported in LavaStream. These are broadly divided into 6 categories of functions as described below.

## Date and Time functions

### Date

The DATE function returns the current date in Julian format

```
DateVar := DATE();
```

#### Parameters

None.

#### Return values

The current Julian date, a numeric value.

#### Remarks

As the returned date is in numeric format, representing the number of days since the year 300 CE, date arithmetic may be performed by simple subtraction or addition.

The return value may be converted to a text representation using the [Format](#) function.

The return variable may be specified either as an INTEGER or as a DATE. If specified as a DATETIME, only the date portion (the integer portion) will be set - the time portion (the fractional portion) will be left blank, i.e. midnight..

### DateFormat

The DATEFORMAT function sets the default date format for conversion purposes.

```
DATEFORMAT(FormatX.FORMAT_S_DATE_4);
DateVar := StringVar;
```

#### Parameters

Date format constant

#### Return values

None.

#### Remarks

After setting the default date format, when assigning string variables into a DATE - type variable, the string is interpreted to be formatted in the specified way. Provided the string has this format, it will be converted into a Julian date corresponding to the string date.

#### See also

[Specific Typecasting \(Dates\)](#)

### Day

The DAY function returns the day of the month from a date or datetime variable containing a valid date

```
DATE (DateVar) ;
```

**Parameters**

DateVar : A DATE or DATETIME variable containing a valid julian date, typically returned by the DATE or DATETIME functions.

**Return values**

The day of the month represented by the nominated date parameter.

**Remarks**

The return value is numeric, and if used in string construction will yield a single digit if the day is less than 10. If more formal string date construction is required, see the FORMAT function.

## Hour

The HOUR function returns the hour of the day in 24 hour format from a time or datetime variable containing a valid time.

```
HOUR (TimeVar) ;
```

**Parameters**

TimeVar : A TIME or DATETIME variable containing a valid Lava time, typically returned by the TIME or DATETIME functions.

**Return values**

The hour of the day in 24 hour format represented by the nominated time parameter.

**Remarks**

The return value is numeric, and will range from 0 through 23.

## Minute

The MINUTE function returns the minute of the hour from a time or datetime variable containing a valid time.

```
MINUTE (TimeVar) ;
```

**Parameters**

TimeVar : A TIME or DATETIME variable containing a valid Lava time, typically returned by the TIME or DATETIME functions.

**Return values**

The minute of the hour represented by the nominated time parameter.

**Remarks**

The return value is numeric, and will range from 0 through 59.

## Month

The MONTH function returns the month of the year from a date or datetime variable containing a valid date

```
MONTH (DateVar) ;
```

**Parameters**

DateVar : A DATE or DATETIME variable containing a valid julian date, typically returned by the DATE or DATETIME functions.

### Return values

The month of the year represented by the nominated date parameter.

### Remarks

The return value is numeric, and if used in string construction will yield a single digit if the month is less than 10. If more formal string date construction is required, see the `FORMAT` function.

## Second

The `SECOND` function returns the second of the minute from a time or datetime variable containing a valid time.

```
SECOND (TimeVar) ;
```

### Parameters

TimeVar : A `TIME` or `DATETIME` variable containing a valid Lava time, typically returned by the `TIME` or `DATETIME` functions.

### Return values

The second of the minute represented by the nominated time parameter.

### Remarks

The return value is numeric, and will range from 0 through 59.

## TimeFormat

The `TIMEFORMAT` function sets the default time format for conversion purposes.

```
TIMEFORMAT (FormatX.FORMAT_S_TIME_4) ;  
TimeVar := StringVar ;
```

### Parameters

Time format constant

### Return values

None.

### Remarks

After setting the default time format, when assigning string variables into a `TIME` - type variable, the string is interpreted to be formatted in the specified way. Provided the string has this format, it will be converted into a Lava time corresponding to the string time.

### See also

[Specific Typecasting \(Times\)](#)

## Year

The `YEAR` function returns the current year from a date or datetime variable containing a valid date

```
MONTH (DateVar) ;
```

### Parameters

DateVar : A `DATE` or `DATETIME` variable containing a valid julian date, typically returned by the `DATE` or `DATETIME` functions.

### Return values

The year represented by the nominated date parameter.

### Remarks

The year is returned in complete form, in other words if the date variable contains the Julian date for (say) 1 January 2007, the YEAR function will return the numeric value 2007.

### Time

The TIME function returns the current time in fractional format.

```
TimeVar := TIME();
```

### Parameters

None.

### Return values

The current time, represented as a fraction of a day.

### Remarks

As the time is represented in a simple numeric format representing a fraction of a day, arithmetic may be performed where time calculation is required.

The return value may be converted to a text representation using the [Format](#) function.

### Datetime

The DATETIME function returns the current date and time in a integer (date) and fraction (time) format.

```
DateTimeVar := DATETIME();
```

### Parameters

None.

### Return values

A DateTime variable, represented in floating point. The integer portion is the Julian date, and the fractional portion is the number of milliseconds since midnight, divided by the number of milliseconds in a day..

### Remarks

As the date and time are in simple numeric format, arithmetic may be performed. The time portion is represented as a fractional day (0 represents midnight and 0.999999988426 represents 1 millisecond before the next midnight) allowing arithmetic both on days and on time; in other words a given date + 12 hours added to (say) a number of 1.5, representing 1 day and 12 hours, will yield exactly 3 days at midnight past the first date, as the time portions will add correctly to 1 additional day.

### MsDate

The MSDATE function returns a Julian date representing the interpreted date as represented in a Quad (8-byte integer) value containing a MS SQL Timestamp value.

```
MSDATE(QuadVariable);
```

### Parameters

QuadVariable : A Quad variable containing an MS SQL Timestamp.

### Return values

The Julian date representation of the Timestamp provided. This is the native date format in LavaStream and in the Lava database.

## Manual Scope and Target Audience

### **Remarks**

The Timestamp value presented would be obtained through an ODBC select from an MS SQL database, as well as some other commercial databases. If selected into a LavaStream variable, the corresponding field in the ROWTYPE structure should be specified of type QUAD.

### **See also**

[Specific Typecasting \(Dates\)](#)

## Variable functions

These functions operate on variables, and either return a derived value or modify the specified variable.

### **BITAND**

Future provision - bitwise AND function.

### **BITOR**

Future provision - bitwise OR function.

### **BITXOR**

Future provision - bitwise XOR function.

### **Clear**

The CLEAR procedure accepts as a single parameter any variable of any type, and clears (nulls) the variable.

```
CLEAR (Variable) ;
```

#### **Parameters**

Variable : Any variable

#### **Return values**

None

#### **Remarks**

The variable is clear (null, empty) after execution of the function. Any values contained in the variable are lost.

### **Dec**

The DEC procedure accepts either a single parameter, being a variable, or two parameters, a variable and a decrement value. The variable is decremented according to the parameters specified.

Form 1 :

```
DEC (Variable) ;
```

Form 2 :

```
DEC (Variable, Value) ;
```

#### **Parameters**

Form 1 :

Variable : Any variable

Form 2 :

Variable : Any variable

Value : Any numeric value or variable containing a numeric value

#### **Return values**

None

#### **Remarks**

In form 1, the nominated variable is decremented by one (1). In form 2, the nominated variable is decremented by the numeric value specified, or the numeric value of the variable specified as the second parameter.

### **Even**

The EVEN function returns a boolean status depending on the odd / even character of the variable specified.

```
EVEN (Variable) ;
```

### Parameters

Variable : Any variable

### Return values

A boolean value, TRUE or FALSE, indicating whether the numeric value of the specified variable is even, as opposed to odd.

### Remarks

The variable is interpreted as an integer numeric value. Other variable forms are first converted to integer before evaluation.

## Inc

The INC procedure accepts either a single parameter, being a variable, or two parameters, a variable and an increment value. The variable is incremented according to the parameters specified.

Form 1 :

```
INC (Variable) ;
```

Form 2 :

```
INC (Variable, Value) ;
```

### Parameters

Form 1 :

Variable : Any variable

Form 2 :

Variable : Any variable

Value : Any numeric value or variable containing a numeric value

### Return values

None

### Remarks

In form 1, the nominated variable is incremented by one (1). In form 2, the nominated variable is incremented by the numeric value specified, or the numeric value of the variable specified as the second parameter.

## Entier

The ENTIER function returns the integer portion of a specified numeric variable.

```
ENTIER (Variable) ;
```

### Parameters

Variable : Any numeric variable.

### Return values

The integer portion of the number. If the number is already an integer, the number is returned unchanged. For floating point numbers, the number is truncated - see ROUND for a rounded return.

### Remarks

If a string variable is passed, an attempt will be made to convert the string to a number. If this attempt fails, the

function returns 0.

## Length

The LENGTH function returns the length of the variable nominated.

```
LENGTH (Variable) ;
```

### Parameters

Variable : Any variable.

### Return values

The length of the variable. The exact meaning of the return depends on the type of the variable.

Simple variables	:	The length in bytes of the variable. For example, the length of an integer variable will return 4.
String variables	:	The length of the string (in bytes or characters) contained in the variable to the terminating null character. In other words, the length of the actual string contained in a variable, not the length of the variable.
File variables	:	The length in bytes of the nominated file.
Table variables	:	The number of rows in the table.
Row type variables	:	Undefined at present.
Lava row variables	:	The number of bytes in the row definition.

### Remarks

Row type variables present a problem, as the exact length of a row type variable is not defined. Each variable within a row type which is of type string is actually variable length - as a result, determining the actual length of the row is not possible.

## Odd

The ODD function returns a boolean status depending on the odd / even character of the variable specified.

```
ODD (Variable) ;
```

### Parameters

Variable : Any variable

### Return values

A boolean value, TRUE or FALSE, indicating whether the numeric value of the specified variable is odd, as opposed to even.

### Remarks

The variable is interpreted as an integer numeric value. Other variable forms are first converted to integer before evaluation.

## Round

The ROUND function rounds a number to the nearest integer value.

```
ROUND (Variable) ;
```

### Parameters

Variable : Any numeric variable.

**Return values**

The closest (rounded) integer value for the number. For integer parameters, the number is returned unchanged.

**Remarks**

If a string variable is passed, an attempt will be made to convert the string to a number. If this attempt fails, the function returns 0.

**Abs**

The ABS function returns the absolute value of a specified numeric value.

```
ABS (Variable) ;
```

**Parameters**

Variable : Any numeric variable.

**Return values**

The absolute value of the specified number.

**Remarks**

If the specified variable is a string, an attempt will be made to convert the string to a number, after which the absolute value is returned.

## String functions

### Char

The CHAR function returns the character for a specified ordinal value.

```
CHAR(Variable);
```

#### Parameters

Variable : Any numeric variable.

#### Return values

A single character representing the ordinal value specified.

#### Remarks

If the variable is a string, an attempt will be made to convert it to a number. If this succeeds, the number will be interpreted as an ordinal. If the specified number evaluates to 0 or less, or a number greater than 255, a null character is returned.

### IPfromString

The IPFROMSTRING function returns a numeric IP address from a valid string representing an IP value

```
IPFROMSTRING(StringVariable);
```

#### Parameters

StringVariable : Any string variable containing a valid IP address

#### Return values

A numeric representation of an IP address (in binary format) - the return value should be stored in an INTEGER variable.

#### Remarks

The string variable must contain a valid IP address, such as "192.168.1.1" (without the quotation marks). If the formatting of the IP address is not valid, the function will return an invalid binary IP value.

### Lower

The LOWER function forces a string to lowercase characters

```
LOWER(StringVariable);
```

#### Parameters

StringVariable : Any string variable.

#### Return values

The entire string converted to lowercase characters.

#### Remarks

The string variable specified is converted to lowercase characters.

**Ord**

The ORD function returns the ordinal value for a character.

```
ORD(CharacterVariable);
```

**Parameters**

CharacterVariable : An indexed string variable

**Return values**

The ordinal (ASCII) value for the first character.

**Remarks**

If the variable is non-indexed, the ordinal value of the first character in the string will be returned. If the variable is not a string variable, the variable will be converted to a string, and the first character's ordinal will be returned.

**ReplaceChar**

The REPLACECHAR function replaces all occurrences of a given character in the string with a specified replacement character

```
REPLACECHAR(StringVariable, SearchChar, ReplacementChar);
```

**Parameters**

StringVariable : Any string variable.  
 SearchChar : A character to be replaced in the string  
 ReplacementChar : The required replacement character

**Return values**

None.

**Remarks**

The string variable specified is modified by the replacement as specified. If the search character is not matched, no change occurs.

**Slicing**

String slicing allows extraction of a substring from a variable, or replacement of a substring in a target variable.

```
StringA := StringB[2..4];
StringA[3..5] := StringB;
StringA[3..4] := StringB[0..3];
StringA[4..4] := StringB;
StringA[4] := StringB[2];
StringA[4] := StringB;
```

From the above examples it is clear that slicing is not a conventional function, but operates in-line during evaluation of source and target variables.

**Remarks**

If the target variable is specified as in the last two examples above, only a single character in the target variable is replaced regardless of the length of the source string (StringB in this case).

If the target variable is specified as a slice (including the specification of [4..4] above, which although this specifies only one character is still evaluated as a slice) any number of characters (fewer or more than the slice) may be inserted as a replacement.

For examples of operation, see [String Assignment](#)

### StringFromIP

The STRINGFROMIP function returns a string representing the provided numeric (binary) IP address

```
STRINGFROMIP(IntegerVariable);
```

#### Parameters

IntegerVariable : An integer variable containing a valid numeric (binary) IP address.

#### Return values

A string representation of an IP address.

#### Remarks

The provided integer variable must contain a valid binary IP address - if this is the case, the function will return a conventional string representation, such as "192.168.1.1" (without the quotation marks).

### StringPos

The STRINGPOS function searches a nominated string (the first parameter) for a given string fragment (the second parameter) and returns the offset at which the fragment was found or -1 if not found.

Form 1 :

```
STRINGPOS(StringVariable, StringFragment);
```

Form 2 :

```
STRINGPOS(StringVariable, StringFragment, StartPos);
```

#### Parameters

Form 1 :

StringVariable : Any string variable. This parameter may also be a constant string.

StringFragment : Any string constant or string variable containing a string fragment

Form 2 :

StringVariable : Any string variable

StringFragment : Any string constant or string variable containing a string fragment

StartPos : A numeric value or variable containing a numeric value representing the start position for the search

#### Return values

The position at which the string fragment was found in the nominated string. The first character of the string is numbered zero (0), thus the lowest valid return value is 0. If the fragment is not found, the function returns -1.

#### Remarks

In form 1, the search begins at the first character (character 0) of the nominated string. In form 2, the search begins at the nominated start position - if the start position evaluates to 0, this is exactly equivalent to form 1.

### TrimAll

The TRIMALL function strips the leading and trailing whitespace from a string value.

```
TRIMALL(StringVariable);
```

#### Parameters

StringVariable : Any string variable.

### Return values

The content of the string with all leading and trailing whitespace removed.

### Remarks

The string variable or value specified is stripped of all leading and trailing whitespace. This includes spaces, tabs, carriage return and linefeed characters.

## TrimLeft

The TRIMLEFT function strips the leading whitespace from a string value.

```
TRIMLEFT (StringVariable) ;
```

### Parameters

StringVariable : Any string variable.

### Return values

The content of the string with all leading whitespace removed.

### Remarks

The string variable or value specified is stripped of all leading whitespace. This includes spaces, tabs, carriage return and linefeed characters.

## TrimRight

The TRIMRIGHT function strips the trailing whitespace from a string value.

```
TRIMRIGHT (StringVariable) ;
```

### Parameters

StringVariable : Any string variable.

### Return values

The content of the string with all leading spaces removed.

### Remarks

The string variable or value specified is stripped of all trailing whitespace. This includes spaces, tabs, carriage return and linefeed characters.

## Upper

The UPPER function forces a string to uppercase characters.

```
UPPER (StringVariable) ;
```

### Parameters

StringVariable : Any string variable.

### Return values

The entire string converted to uppercase characters.

### Remarks

The string variable specified is converted to uppercase characters.

## Length

## Manual Scope and Target Audience

The LENGTH function returns the length of a nominated string. See [Length](#) for detailed information.

## Lava Database functions

The following functions provide dedicated interfaces to the Lava database. It is possible in almost all cases to achieve the same result using SQL commands, but the function is in most cases faster and generally easier to use. Also, the function is validated at compile time, avoiding unnecessary errors at run-time due to typing or syntactical errors which may occur in an embedded SQL statement.

### AutoCommit

The AUTOCOMMIT function enables Autocommit mode on the current session. This means that no transaction frame will be created for database update commands, and that the Rollback and Commit functions become non-functional for the session.

```
AUTOCOMMIT ( ) ;
```

#### Parameters

None. The session is implied (as the current session).

#### Return values

None.

#### Remarks

The current session becomes Autocommit, i.e. transaction frames are disabled.

Subsequent updates to the database will process much faster, as less processing is required especially if very large updates or bulk inserts (many rows) are performed.

For some types of batch processes, Autocommit mode is more appropriate as the possibility of rollback does not exist. In these cases, switching to Autocommit mode is recommended, especially where very large numbers of rows are to be processed.

### Clear

The CLEAR function, when applied to a Lava table row, has the result of deleting the specified row in the table.

```
CLEAR (TableVar [RowIndex] ) ;
```

#### Parameters

TableVar	:	A variable of type TABLE which specifies a valid Lava table, or a LAVA table constant.
RowIndex	:	The RowIndex is specified as an index to the table, thus qualifying a row in the table.

#### Return values

None.

#### Remarks

The table specified must be valid, and the session must have adequate permissions to delete rows from the table. The row index must specify a valid row in the table.

### ColumnLabel

The COLUMNLABEL function returns the name (label) of a nominated column.

```
StringVar := COLUMNLABEL (TableConst, ColumnSequence) ;
```

### Parameters

TableConst	:	The table ID for a Lava table - typically specified as LAVA.SchemaName.TableName
ColumnSequence	:	The sequence of the column required, which may either be specified as a number (column sequences are 1-based) or a column specifier, formatted as LAVA.SchemaName.TableName.ColumnLabel

### Return values

The label (string) of the nominated column.

### Remarks

The table specified must be valid, and the column sequence specified must be valid for the nominated table.

## ColumnLength

The COLUMNLENGTH function returns the length in bytes of a nominated column.

```
Length := COLUMNLENGTH (TableConst, Row_id, ColumnSequence) ;
```

### Parameters

TableConst	:	The table ID for a Lava table - typically specified as LAVA.SchemaName.TableName
Row_id	:	The row for which the column length is to be returned. May be 0 if the column specified is not a variable length column.
ColumnSequence	:	The sequence of the column required, which may either be specified as a number (column sequences are 1-based) or a column specifier, formatted as LAVA.SchemaName.TableName.ColumnLabel

### Return values

The length in bytes of the nominated column. If the column is a variable length column (such as a VARSTRING) the length is specific to the row nominated.

### Remarks

The table specified must be valid, and the column sequence specified must be valid for the nominated table.

In the case of variable length columns, the row\_id specified must nominate a valid row.

For variable length columns, the length returned is the length of the actual data stored in the column. If the column contains no extended data, the length returned is the base length of the column - i.e. the length of the constant portion of the column as stored in the row definition.

## Commit

The COMMIT function issues a commit request on the current session.

```
COMMIT () ;
```

### Parameters

None

### Return values

None.

**Remarks**

Any pending transaction frame is committed. If no transactions are pending, the function has no effect.

**Createtable**

The CREATETABLE function creates a Lava table according to parameters specified. An optional third parameter specifies creation options (table attributes).

Form 1 :

```
TableVar := CREATETABLE (TableName, RowtypeVariable);
```

Form 2 :

```
TableVar := CREATETABLE (TableName, RowtypeVariable, Attributes);
```

**Parameters**

Form 1 :

- TableName : A string or string variable specifying the desired name of the table.
- RowtypeVariable : A row type variable specifying the format (column layout) of the table

Form 2 :

- TableName : A string or string variable specifying the desired name of the table.
- RowtypeVariable : A row type variable specifying the format (column layout) of the table
- Attributes : One or more specific attributes to be applied to the created table

**Return values**

The table id of the created table.

**Remarks**

If the table creation fails, the function returns 0. Table creation will fail if a table by the same name already exists in the default schema for the session, or if the user does not have adequate privilege to create tables in this schema.

If Form 1 of the command is used, the attributes of the table created default to Virtual and Raw - this provides a table which is the closest approximation to an array, with no spurious columns.

The allowable attributes are provided in the FormatX module.

Attributes are :

SQL_OBJECT_TABLE	Default - the created object is a table
SQL_OBJECT_VIRTUAL	The created table is virtual (a memory table). No physical table is created, and all information stored in the table is transitory (lost on every dismount). If this parameter is omitted, the table is created as a physical table, which will allow permanent storage of data.
SQL_OBJECT_RAW	The table is "raw" - no version information is maintained. Deletion of rows is not supported.
SQL_OBJECT_PERSISTENT	Only applicable to virtual tables created on the server. The table survives dismount / mount cycles. All information contained in the table is still lost on dismount.
SQL_OBJECT_FRAMED	The table supports transaction frames. This attribute requires that the first field in the specified row type is of type version.

SQL_OBJECT_LOCAL	The table is created on the client. The table will only exist for the current session - it will be lost on dismount.
SQL_OBJECT_SERVER	The table is created on the server. If the table is virtual persistent, or physical, it will survive dismount / mount cycles.

### DropIndex

The DROPINDEX function drops (deletes) the nominated index on a Lava table.

```
DROPINDEX (Table, Column);
```

#### Parameters

Table	:	The Object ID for the required Lava table
Column	:	An optional parameter specifying the column sequence of the column on which the required index is based

#### Return values

None.

#### Remarks

The user must have sufficient permission to drop the specified column.

If the Column parameter is specified, this should be the column sequence of a valid column for the table. If no index exists for the column specified, the function will do nothing.

If the Column parameter is omitted, all indexes for the nominated table will be dropped.

### Droptable

The DROPTABLE function drops (deletes) a Lava table.

```
DROPTABLE (TableVar);
```

#### Parameters

TableVar	:	A Table variable specifying the table to be dropped.
----------	---	--

#### Return values

None.

#### Remarks

The user must have sufficient permission to drop the specified table.

If a table is dropped, there is no way to recover the data stored in the table. Dropping a table should typically be used only on temporary tables.

### FindClose

The FindClose function terminates a find and releases all allocated resources.

```
FINDCLOSE (Seek_id);
```

#### Parameters

Seek_id	:	Seek identifier - returned by FINDROW.
---------	---	--

**Return values**

None.

**Remarks**

This function frees all resources allocated by the FINDROW function. Once a FINDCLOSE has been issued on a given seek identifier, the identifier can no longer be used in seek operations until appropriate initialization (such as FINDROW) has been issued to establish new search parameters.

**See also**

[FindRow](#), [FindColumn](#), [FindNext](#), [FindTable](#)

**FindColumn**

The FINDCOLUMN function allows the search criterion for a single column to be added to a multi-column search on a Lava table.

```
FINDCOLUMN (Seek_id, ColumnSequence, Condition, ColumnValue);
```

**Parameters**

Seek_id	:	An integer variable which identifies a seek identifier which was established using the FINDTABLE command. This identifier is used to specify further seek columns, perform the initial seek, find further matches for the same seek, and to close the seek and free associated resources.
ColumnSequence	:	The sequence of the column on which the search is to be performed. Columns are numbered 1-based (starting at 1).
Condition	:	The condition to be applied to the column data when seeking for the specified column value. Valid conditions are EQ, LT, LEQ, GT, GEQ, NEQ.
ColumnValue	:	The value for which the search is to be performed.

**Return values**

None

**Remarks**

The Seek\_id specified must have been initialized through the FINDTABLE command, which specifies the table on which the seek is to be performed.

Multiple searches may be active simultaneously - since each search operates off a unique Seek ID, it is even possible to have more than one search on one table at any time.

It is important to issue a FINDCLOSE when the search has been terminated, as a search consumes resources and these resources will add up if many searches are left open.

**Example**

The following code sequence would initiate a multi-column seek :

```
FINDTABLE(Seek_id, LAVA.Design.IDE_GridControlColumns);
FINDCOLUMN(Seek_id, LAVA.Design.IDE_GridControlColumns.Grid_id,
           EQ, NodeEntry.Property_id);
FINDCOLUMN(Seek_id, LAVA.Design.IDE_GridControlColumns.SortSequence,
           GEQ, 1);
Column_id := FINDROW(Seek_id);
WHILE Column_id # 0 DO
(* Required processing goes here *)
  Column_id := FINDNEXT(Seek_id);
```

```
END;
```

In the above example the Grid\_id column is specified as being equal to the variable NodeEntry.Property\_id, and the SortSequence column is specified as greater than or equal to 1. All columns returned from the FINDROW / FINDNEXT calls will match these criteria - in addition, where an inequality is specified, the results will be sorted ascending for Greater inequalities and descending for Less inequalities.

Note that in the above example the FINDROW call specifies only the seek identifier - all seek parameters have already been stored in the seek identifier using the FINDTABLE / FINDCOLUMN calls.

**See also**

[FindRow](#), [FindNext](#), [FindClose](#), [FindTable](#)

## FindNext

The FINDNEXT function locates the next match after a FINDFIRST has been called.

```
RowVar := FINDNEXT(Seek_id);
```

**Parameters**

Seek\_id : The seek identifier returned by FINDROW.

**Return values**

The row id of the next hit. If no further occurrence of the search is found, the returned row id is zero.

**Remarks**

In order for the FINDNEXT routine to yield sensible results, a call to FINDROW must be executed first.

FINDNEXT obeys the seek parameters specified for the given seek identifier, either as specified in the FINDROW call, or as specified by a FINDTABLE / FINDCOLUMN set.

**See also**

[FindRow](#), [FindColumn](#), [FindClose](#), [FindTable](#)

## FindRow

The FINDROW function provides a single-column search on a Lava table. A binary search is performed to locate the first hit.

FINDROW is presented in two forms :

Form 1 :

```
RowVar := FINDROW(Seek_id, TableVar, ColumnSequence, ColumnValue);
```

Form 2 :

```
RowVar := FINDROW(Seek_id);
```

**Parameters**

Form 1 :

Seek\_id : An integer variable which receives a seek identifier, used subsequently to find further matches for the same seek, and to close the seek and free associated resources.

TableVar : A valid Lava table.

ColumnSequence : The sequence of the column on which the search is to be performed. Columns are numbered 1-based (starting at 1).

ColumnValue : The value for which the search is to be performed.

Form 2 :

Seek\_id : An integer variable which receives a seek identifier, previously

## Manual Scope and Target Audience

initialized by calls to `FINDTABLE / FINDCOLUMN`, used subsequently to find further matches for the same seek, and to close the seek and free associated resources.

### Return values

The row id of the first row matching the search. This row id may be used to access Lava table rows as the index to the table specifier.

### Remarks

In form 1, the `FINDROW` is used to perform an equality seek on a single column of the nominated table. In form 2, `FINDROW` may be used to perform a seek on multiple column conditions for a table, where both equality and inequality conditions are supported. Form 2 may also be used to perform a single column seek where an inequality condition is required.

When using form 1, the specified column may not be of type boolean or byte - searches are not supported on boolean or byte columns.

Multiple searches may be active simultaneously - since each search operates off a unique Seek ID, it is even possible to have more than one search on one table at any time.

It is important to issue a `FINDCLOSE` when the search has been terminated, as a search consumes resources and these resources will add up if many searches are left open.

### When to use FindRow

The `FindRow` function is presented in order to find specific rows or sets of rows in a Lava table. Conventionally, this form of selection is performed using a SQL Select command, and this is also an option in the Lava database. However, it is sometimes more convenient to perform the location of matching rows through direct functionality, providing row Ids which can be used immediately for row-level operations.

Typical instances where a row seek may be used in preference to a SQL Select will include cases where the SQL select to derive the required matching data set will be too complex, and further processing will be required in order to check the results. In cases such as this, it is perhaps preferable to perform the entire seek operation at row level, perhaps yielding a result easier to program and test.

### See also

[FindColumn](#), [FindNext](#), [FindClose](#), [FindTable](#)

## FindTable

The `FINDTABLE` procedure specifies the table to be used in subsequent seek operations for a given seek identifier. `FINDTABLE` is the first and initializing call for a multiple-column seek on a table.

```
FINDTABLE (Seek_id, TableVar);
```

### Parameters

Seek_id	:	The seek identifier to be initialized.
TableVar	:	A specification of a Lava table on which a seek is to be performed. This may be either through a <code>LAVA.SCHEMA.TABLE</code> specification, or a <code>TABLE</code> - type variable which specifies a Lava table such as a result set returned from the <code>ODBC_SQL</code> or <code>SQL</code> functions.

### Return values

None. The Seek\_id identifier is initialized.

#### Remarks

This command sets the table on which a seek is to be performed using the nominated seek identifier. For an example of a multi-column seek, see the [FINDCOLUMN](#) command.

#### See also

[FindRow](#), [FindColumn](#), [FindNext](#), [FindClose](#)

### FreeRow

The FREEROW function returns the first free row in the nominated table, that is the first row available for a new entry.

```
RowVar := FREEROW (TableVar);
```

#### Parameters

TableVar : The table for which the free row is requested.

#### Return values

The row id for the first available row in the table.

#### Remarks

If the table is flagged as *reclaim deleted*, the row returned may be a previously deleted row within the current extent of the table. If not, or if the table has no deleted rows, the row returned will be at the end of the table.

### GetColumn

The GETCOLUMN function retrieves the content of a single column for a nominated row in a data table.

```
GETCOLUMN (TableSpec, Row_id, ColumnSpec, ColumnValue);
```

#### Parameters

TableSpec : A specification for the table required  
 Row\_id : The row for which the column is required  
 ColumnSpec : Specification of the column from which the value is required  
 ColumnValue : A variable to receive the value of the column

#### Return values

None.

#### Remarks

If the request fails, the lava status will be set to a non-zero value. This may be checked through use of the LAVASTATUS function.

The request will fail if the user has insufficient access rights for the table, or the row is invalid (deleted / unused).

Variable length columns can only be retrieved in full through use of this function. Retrieving a row in the table will only retrieve the default length portion of a variable length column - the remainder will be ignored as there is no provision for the additional data in the row structure.

#### See also

[SetColumn](#)

**LavaError**

The LAVAERROR function returns a description for a provided return code.

```
ErrorString := LAVAERROR(ReturnCode);
```

**Parameters**

ReturnCode : The return code for which a description is required.

**Return values**

A string description for the return code.

**Remarks**

Return codes are returned by the LAVASTATUS function, which provides the return code resulting from the previous Lava or ODBC access function. Where possible, the LAVAERROR function will find and return a description for such a return code.

**LavaStatus**

The LAVASTATUS function returns the last return code resulting from the previous call to either a Lava database or an ODBC function call.

```
ReturnCode := LAVASTATUS();
```

**Parameters**

None.

**Return values**

The most recent return code resulting from a Lava or ODBC function call.

**Remarks**

If the most recent call to a database function was successful, the returned code will be zero.

**OpenSession**

The OPENSESSION function creates a new session to a Lava database.

```
Session := OPENSESSION(User, Password, Server);
```

**Parameters**

User : A valid user name on the server  
 Password : The password for the user account  
 Server : The name or IP address of the Lava server

**Return values**

A session id for the new session, or zero if creation of the session fails.

**Remarks**

The new session does not automatically become the active session. In order to switch to a given session, the SETSESSION function can be used.

**RenameTable**

The RenameTable function renames a Lava table (specified in the form of a Table ID variable) to a given name.

```
RENAMETABLE(TableVar, NewName);
```

### Parameters

TableVar	:	A Table variable specifying the table to be dropped.
newName	:	The desired name for the nominated table

### Return values

None.

### Remarks

The user must have sufficient permission to rename the specified table.

## ReserveRows

The RESERVEROWS function reserves the nominated number of rows for the specified table, in order to do bulk inserts into a table.

```
RESERVEROWS (TableSpec, RowCount);
```

### Parameters

TableSpec	:	A valid Lava table id, specified either as a numeric variable containing a valid table id, or as a table constant formatted as LAVA.SchemaName.TableName.
RowCount	:	A numeric value specifying a valid number of rows to be reserved.

### Return values

None. Success or failure may be determined by using the LAVARC function.

### Remarks

The nominated table must resolve to a valid Lava table, on which the user has rights to add rows.

The row count nominated must be valid - zero or negative numbers are invalid, as are very large numbers (up to several million rows should not present a problem, although larger numbers will take longer to process).

On successful reservation, processing of bulk inserts into the nominated table will process much faster, as repeated reservation of the default number of rows from the server (typically 10 on the majority of tables) will be obviated.

## Rollback

The ROLLBACK function performs a rollback on any pending transaction for the current session.

```
ROLLBACK ();
```

### Parameters

None.

### Return values

None.

### Remarks

If no pending transaction exists, the ROLLBACK function will have no effect.

## SetSession

The SETSESSION function is used to switch the active session. An OPENSESSION function must be called successfully in order to obtain a valid alternative session.

```
SETSESSION (Session) ;
```

The SetSession function allows the current session to be set to another session than the default. Such a session would have to be obtained using the OpenSession function.

### Parameters

Session : A valid Lava session id.

### Return values

None.

### Remarks

The SETSESSION function will only succeed if a valid alternative session is provided. If not, the command will have no effect.

## Session

The Session function returns the current session id.

### Parameters

None.

### Return values

The current session id.

### Remarks

This function may be used if it is necessary to temporarily switch to an alternate session - for example, if a batch process is to be performed which must be switched to Autocommit processing. In such a case, the current session may be retrieved by using the Session function and saved in a temporary variable, a new session may be obtained using the OpenSession function, the current session may be switched, the new session can be set to Autocommit, and when the processing is complete the session may be reset to the original default session which has transaction processing enabled. The following code snippet illustrates such a switch.

```
Session_id := SESSION();
TempSession_id := OPENSESSION('UserName', 'Password', 'Server');
SETSESSION(TempSession_id);
AUTOCOMMIT();
:
:
SETSESSION(Session_id);
```

## SetColumn

The SETCOLUMN function sets (writes) the content of a single column for a nominated row in a data table.

```
SETCOLUMN(TableSpec, Row_id, ColumnSpec, ColumnValue, Length);
```

### Parameters

TableSpec	:	A specification for the table required
Row_id	:	The row for which the column is to be set
ColumnSpec	:	Specification of the column to be set
ColumnValue	:	A variable or other specification of the required value of the column
Length	:	An optional parameter specifying the length of the required data. If omitted, the full length of the nominated value is written to the column.

### Return values

## Manual Scope and Target Audience

None.

### Remarks

If the request fails, the lava status will be set to a non-zero value. This may be checked through use of the LAVASTATUS function.

The request will fail if the user has insufficient access rights for the table, or the row is invalid (deleted / unused).

Variable length columns may only be written in full through use of this function. Writing / adding a row in the table through the conventional array access to the table will only set the default length portion of a variable length column - since there is no way to specify any further information in the default row structure of the table, no further information may be written to variable length columns using the array access method.

If the target column is a variable length column, the column value specified may be of any length. The full content of the variable will be written to the column. Note that the full content of such a variable length column can only be retrieved through use of the GETCOLUMN function.

For variable length columns the optional final parameter (length) may be specified if the full length of the value is not required. If specified, only this number of bytes (characters) will be written to the column. If omitted (the default operation) the full length of the value will be written to the column.

Any column may be written to a row in a table through this function - even non-variable length columns.

For non-variable length columns (fixed length values such as integers, floats, dates...) the last parameter is ignored - the length of the column dictates the number of bytes written, regardless of the length parameter.

### See also

[GetColumn](#)

## Sql

The Lava SQL function executes SQL commands on the Lava server.

```
ResultTable := SQL(Command);
```

### Parameters

Command : A valid SQL command.

### Return values

The id of the resultant table, if any.

### Remarks

When *select* commands are executed on a Lava database, the database creates a result set which is a Lava data table. In such a case, the table id of this result table is returned on completion.

If the SQL command fails, or if the command does not result in a table, the result table id will be zero.

## Tablecolumns

The TABLECOLUMNS function returns the number of columns in the specified table.

```
Columns :=TABLECOLUMNS (TableVar);
```

### Parameters

TableVar : A valid Lava table id.

**Return values**

The number of columns in the table.

**Remarks**

If the table is invalid, the function returns a column count of zero.

**TableName**

The TableName function returns the name (in text) of the nominated table.

```
StringVar:=TABLENAME (TableSpec);
```

**Parameters**

TableSpec : A valid Lava table id, specified either as a numeric variable containing a valid table id, or as a table constant formatted as LAVA.SchemaName.TableName.

**Return values**

The name (in text) of the table

**Remarks**

If the table is invalid, the function returns an empty string.

**Tablerows**

The TABLEROWS function returns the number of rows in the specified table.

```
Rows := TABLEROWS (TableVar);
```

**Parameters**

TableVar : A valid Lava table id.

**Return values**

The number of rows in the table.

**Remarks**

The number of rows returned is not necessarily the number of active (non-deleted) rows. In order to support array-like processing of tables, the row count returned actually represents the row id of the highest-numbered active row in the table. If rows prior to this have been deleted, the row count will not be the same as the number of active rows.

In the example below, the standard technique for full table processing is shown.

```
TYPE
  UserType = ROWTYPE
  id       : INTEGER;
  User     : STRING[50];
  Schema   : STRING[50];
END;

PROCEDURE TestSelect();
VAR
  Result   : TABLE;
  SQLcommand : STRING;
  Index    : INTEGER;
```

```

User      :  UserType;
rc       :  INTEGER;
BEGIN
(* Get the user list from the db *)
SQLcommand := 'select ' &
               'id, user_name, sys_schemas.schema_name ' &
               'from ' &
               'system.sys_users, system.sys_schemas ' &
               'where ' &
               'sys_users.schema_id = sys_schemas.id';

Result := SQL(SQLcommand);

(* Process each valid row *)
FOR Index := 1 TO TABLEROWS(Result) DO
  Row := Result[Index];
  rc := LAVASTATUS();
  IF rc # 0 THEN
    CYCLE;
  END;
  (* process row here - dummy output follows *)
  OUTPUT(User.id & ' ' & User.User & ' ' & User.Schema & 0DX);
END;
END TestSelect;

```

## Translate

The TRANSLATE command uses a user-maintained translation table to translate a given source value to a target (translation) value as specified in the table.

```

rc := TRANSLATE( SourceValue,
                 TargetValue,
                 TableSpec,
                 SourceColumn,
                 TargetColumn);

```

### Parameters

SourceValue	:	The value to be translated. This may be either a constant or variable, in fact any form of specification of the required translation value.
TargetValue	:	A variable to receive the translated value. The variable should be of the appropriate type to receive the content of the target column in the nominated translation table.
TableSpec	:	A valid Lava table id, specified either as a numeric variable containing a valid table id, or as a table constant formatted as LAVA.SchemaName.TableName. The translation table should contain appropriate values to translate the required source values.
SourceColumn	:	Specification of the column containing a list of source values to be translated (in the nominated translation table). The column should be specified either as an integer value containing the column id, or as a LAVA.SchemaName.TableName.ColumnName specification.
TargetColumn	:	Specification of the column containing target values (translated values) for possible source values.(in the nominated translation table). The column should be specified either as an integer value containing the column id, or as a LAVA.SchemaName.TableName.ColumnName specification.

### Return values

A return code indicating whether the nominated source value was found and successfully translated. A return code of 0 indicates successful translation.

### Remarks

The nominated table must exist and be distributed for translation to succeed.

All appropriate source values should be listed in the translation table. Any missing source values will result in failed translation, and the given source value will merely be copied to the target value variable (no translation will be performed).

If the source value is found in the translation table, the corresponding target value in the nominated target column will be copied into the target value variable.

If duplicates of a given source value exist in the translation table, the translation function will respond with the first entry found that corresponds with the nominated source value.

Source values are determined on a case insensitive basis.

## Truncate

The TRUNCATE function truncates the nominated table.

```
TRUNCATE (TableSpec) ;
```

### Parameters

TableSpec	:	A valid Lava table id, specified either as a numeric variable containing a valid table id, or as a table constant formatted as LAVA.SchemaName.TableName.
-----------	---	---

### Return values

None.

### Remarks

If the table is invalid, not distributed or the session has insufficient privilege to truncate the table, the function will fail.

The success or failure of the function may be determined through use of the LAVASTATUS function. Successful execution will result in a LAVASTATUS value of 0.

After successful execution, the table will be empty. All rows in the table will be irretrievably lost. No rollback recovery is possible after a truncate command.

## ODBC Database functions

The ODBC functionality integrated into LavaStream provides the ability to access third-party databases in order to obtain or update data tables.

As LavaStream has the ability to buffer even large amounts of data, it is possible to import data tables directly for purposes of filtering, data transforms or complex update operations - either with a single third-party database or between databases, potentially from different vendors.

The ODBC functions, as with Lava database functions, return error codes via the LAVASTATUS function.

### ODBC\_Close

The ODBC\_CLOSE function closes a current ODBC connection and frees all resources associated with the connection.

```
ODBC_CLOSE (ODBC_connection);
```

#### Parameters

ODBC\_connection : A valid connection returned by the ODBC\_CONNECT function.

#### Return values

None.

#### Remarks

The function will have no effect if the connection specified is not valid.

### ODBC\_Connect

In order to connect to an ODBC source, a successful Connect function must be performed. A valid ODBC DSN must be created using the Windows Data Sources (ODBC) dialog. For documentation on this dialog, the specific third-party ODBC driver documentation should provide further information.

```
ODBC_connection := ODBC_CONNECT (DSNspec);
```

#### Parameters

DSNspec : A valid DSN specification in the Data Sources dialog.

#### Return values

An ODBC\_connection, which may be used for further ODBC functions.

#### Remarks

The DSN specification must be of the form 'DSN=abcd', where **abcd** is the name (for the data source) specified in the Data Sources list.

### ODBC\_Sql

The ODBC\_SQL function allows SQL commands to be executed on the connected ODBC database. Provided the SQL is correctly stated and is valid, the statement will execute and (potentially) deliver result data.

```
ResultTable := ODBC_SQL (ODBC_connection, Command);
```

#### Parameters

ODBC\_connection : A valid connection returned by the ODBC\_CONNECT function.  
Command : A correctly formed SQL command, according to the rules and

## Manual Scope and Target Audience

syntax of the target ODBC database.

### Return values

The Lava Table id of a result table. If the SQL command does not yield any results - either because the command is not intended to yield result data (the case with UPDATE commands, for example) or because of error or lack of matching result rows, the Table id will be zero.

### Remarks

If a table id is returned, a Lava table has been populated with the data results returned by an ODBC select statement. This table may be processed as for any other Lava table, but is virtual (a memory table) and will be lost on dismount.

Note that the SQL must be formatted according to the syntactic rules of the target ODBC database.

## ODBC\_Add

The ODBC\_ADD function is a facility for adding rows to an ODBC target table from within LavaStream.

```
ODBC_Add(ODBC_Connection, TableName, RowTypeVar);
```

### Parameters

ODBC_connection	:	A valid connection returned by the ODBC_CONNECT function.
TableName	:	The name of the table on the target database to which the row is to be added.
RowTypeVar	:	A LavaStream variable of type ROWTYPE.

### Return values

None.

### Remarks

The Add function is exactly equivalent to an SQL *insert into* command. The specified RowType variable is used as the specifier of column names and content. The field names of the variable are used as the column names, and the content of the fields is specified as the insert values. The following example :

```
TYPE
  FormType      = ROWTYPE
  customer      : STRING[100];
  address       : STRING[100];
  district      : STRING[100];
END;
```

:

```
VAR
  Form          : FormType;
  Connect       : INTEGER;
BEGIN
  Connect := ODBC_CONNECT('DSN=DestDBdsn');
  Form.customer := 'John';
  Form.address := '238 Freddie Avenue, Epstom';
  Form.district := 'Far Cotswold';
  ODBC_ADD(Connect, 'FormTable', Form);
```

The above Add command would be the exact equivalent of the SQL command

```
insert into FormTable
  (Customer, address, district)
```

```
values
    ('John', '238 Freddie Avenue, Epstom', 'Far Cotswold')
```

## ODBC\_Update

The ODBC\_UPDATE function is a facility for updating rows in an ODBC target table from within LavaStream.

```
ODBC_UPDATE(ODBC_Connection, TableName, RowTypeVar, RowFilter);
```

### Parameters

ODBC_connection	:	A valid connection returned by the ODBC_CONNECT function.
TableName	:	The name of the table on the target database to which the row is to be added.
RowTypeVar	:	A LavaStream variable of type ROWTYPE.
RowFilter	:	A filter specification (valid for the nominated ODBC connection) which will filter for the row to be updated.

### Return values

None.

### Remarks

The Update function is exactly equivalent to an SQL *update* command. The specified RowType variable is used as the specifier of column names and content. The field names of the variable are used as the column names, and the content of the fields is specified as the update values.

Note that the filter specified must be valid in syntax and effect for the nominated ODBC connection. This filter will be appended to an *update* command, as illustrated in the following example :

```
TYPE
    StockType      = ROWTYPE
    StockCode      : INTEGER;
    Quantity       : INTEGER;
    Description    : STRING[100];
END;
```

```
VAR
    Stock          : StockType;
    Connect        : INTEGER
BEGIN
    Connect := ODBC_CONNECT('DSN=DestDBdsn');
    Stock.StockCode := 1015;
    Stock.Quantity := 23;
    Stock.Description := 'Split pin 25mm';
```

```
ODBC_UPDATE(Connect, 'StockTable', Stock, 'stockcode = 1015');
```

The above LavaStream command will translate as follows :

```
update
    stocktable
set
    stockcode = 1015, quantity = 23, description = 'Split pin 25mm'
where
    stockcode = 1015
```

## Manual Scope and Target Audience

## Text Output and Formatting Functions

### Output

The OUTPUT function is the sole function used to output data to the Apache data buffer. This is used to output both HTML and XML data to client browsers.

```
OUTPUT(StringValue);
```

#### Parameters

StringValue : Any string value or variable.

#### Return values

None.

#### Remarks

The parameter may be constructed through concatenation, and variables may be other than strings - these will be converted to strings before output.

Note that if any of the output values are calculations, these should be placed in parentheses to force early evaluation. For example :

```
OUTPUT('Test ' & (2 * 3 - 4) & ' more');
```

The above example will output the string **Test 2 more**

If the parentheses are omitted, the results will be strange, probably something like **-4 more**. The reason for this is left-to-right evaluation, which will compile the output in the following steps :

<b>Test</b>	( the first string )
<b>Test 2</b>	( Concatenation of 2 with <b>Test</b> )
<b>0</b>	( the result of <b>Test 2</b> evaluated as a number, multiplied by 3 - zero times 3 )
<b>-4</b>	( the result of <b>0 - 4</b> )
<b>-4 more</b>	( -4 concatenated with <b>more</b> )

If a calculation or number results in a floating point value, the default output format will be xxx.dd (two digit fraction). If another output than this is required, the FORMAT command may be used to yield the desired output.

### Format

The FORMAT function provides the ability to convert numeric variables into various forms of string representations, including dates, times and currency forms.

```
StringVar := FORMAT(NumberVar, PrimaryFormat, SecondaryFormat);
```

#### Parameters

NumberVar	:	The number to be formatted
PrimaryFormat	:	The primary format - distinguishes between items such as date, time and other major representations
SecondaryFormat	:	The secondary format - where appropriate specifies detailed format of the output

#### Return values

A string formatted according to the input and parameters provided

**Remarks**

The returned string is formatted firstly according to the primary format specified. This attribute specifies the basic nature of the output.

FORMAT_P_NUMBER	Pure number formatting. Used primarily to format floating-point numbers, but can be used to provide comma delimiting to integers.
FORMAT_P_CURRENCY	Currency formatting. Allows specification of currency symbol and position.
FORMAT_P_ACCOUNTING	Accounting-specific outputs.
FORMAT_P_DATE	Date formats. The secondary format allows selection of a range of date formats representing most standards.
FORMAT_P_TIME	Time formats. The secondary format allows selection of 12 and 24 hour, and subformats.
FORMAT_P_PERCENTAGE	The number is represented as a percentage. Input in the range of 0.01 through 1 are represented as 1% through 100%.
FORMAT_P_SCIENTIFIC	The number is represented in standard scientific format, as a normalised mantissa and exponent.
FORMAT_P_HEXQUAD	A quad-sized (8-byte integer) numeric input is formatted to a eight-character hex number.
FORMAT_P_VDT	A floating point input is formatted into a date and time output.
FORMAT_P_IP	The number is formatted into a 4-segment IP address.
FORMAT_P_BITSET	A integer number is formatted as binary (32-bit) output.
FORMAT_P_BOOLEAN	The number is interpreted as false (0) or true (non-zero).
FORMAT_P_KB_QUAD	The number is output as a binary kilobyte representation, with decimals representing fractional kilobytes.
FORMAT_P_MB_QUAD	The number is output as a binary megabyte representation, with decimals representing fractional megabytes.

The secondary format - optional in some cases - specifies detail qualifying the exact nature of the output.

FORMAT_S_BASE_BINARY	The number is formatted in binary output
FORMAT_S_BASE_HEX	The number is formatted as hexadecimal output
FORMAT_S_NEGATIVE_1	Negatives are represented in conventional -123 format.
FORMAT_S_NEGATIVE_3	Negatives are represented through surrounding parentheses as (123).
FORMAT_S_DATE_1	Date format dd/mm/yy (17/01/64)
FORMAT_S_DATE_2	Date format dd/mm/yyyy (17/01/1964)
FORMAT_S_DATE_3	Date format mm/dd/yy (01/17/64)

FORMAT_S_DATE_4	Date format mm/dd/yyyy (01/17/64)
FORMAT_S_DATE_5	Date format mmm dd, yyyy (Jan 17, 1964)
FORMAT_S_DATE_6	Date format mmmmmmmm dd, yyyy (January 17, 1964)
FORMAT_S_DATE_7	Date format dd mmm yy (17 Jan 64)
FORMAT_S_DATE_8	Date format dd mmm yyyy (17 Jan 1964)
FORMAT_S_DATE_9	Date format yyyy/mm/dd (1964/01/17)
FORMAT_S_TIME_1	Time format hh:mm (14:07)
FORMAT_S_TIME_2	Time format hh:mm:ss (14:07:03)
FORMAT_S_TIME_3	Time format hh:mmXM (2:07PM)
FORMAT_S_TIME_4	Time format hh:mm:ssXM (2:07:03PM)
FORMAT_S_TIME_5	Time format hhmm (1407)
FORMAT_S_TIME_6	Time format hhmmss (140703)

**See also**

[Specific Typecasting \(Dates\)](#)

### XmlHeader

The XMLHEADER function formats an XML header according to the options selected.

```
StringVar := XMLHEADER(VersionReqd, HeaderTag);
```

**Parameters**

- VersionReqd : This boolean parameter specifies whether an XML version header is prefixed to the output.
- HeaderTag : The header tag specifies the text value of the encompassing tag for the section.

**Return values**

A string containing a formatted XML header.

**Remarks**

If the VersionReqd parameter is set to TRUE, the output is prefixed with the version string :

```
<?xml version="1.0" encoding="utf-8" ?>
```

The Header Tag specifies the text in the main tag. The following example illustrates the principle :

```
StringVar := XMLHEADER(FALSE, 'items');
```

results in assignment of the following output to StringVar :

```
<items>
```

### XmlRowdata

The XMLROWDATA function formats the content of a RowType variable into XML format.

```
StringVar := XMLROWDATA(RowTag, RowVar, TypeReqd, LabelReqd);
```

### Parameters

RowTag	:	The text value for the encompassing tag
RowVar	:	The nominated RowType variable
TypeReqd	:	(Optional) A boolean indicator for a type attribute. Default is FALSE.
LabelReqd	:	(Optional) A boolean indicating whether row type fields are output with the field label as the tag (in the case of TRUE) or with a tag of cell (in the case of FALSE). Default is FALSE.

### Return values

A string containing a set of formatted XML tags for the nominated row type.

### Remarks

The RowTag parameter specifies the encompassing tags for the row elements. Each of the row elements is allocated a tag according to the label of the field.

The TypeReqd parameter indicates whether each field is provided an attribute for the type of the field.

If the ROWID property of the RowType variable is non-zero, a ROWID attribute will be appended to the encompassing tag.

The following example illustrates a fully attributed output :

```
Row.ROWID := 235;
Row.Label := 'City';
Row.Value := 'Budapest';
Row.Code := 37;

OUTPUT (XMLROWDATA ('rowData', Row, TRUE);
```

The above results in the following output :

```
<rowData ROWID=235>
  <Label Type=10>City</Label>
  <Value Type=10>Budapest</Value>
  <Code Type=3>37</Code>
</rowData>
```

If the ROWID property were zero, the first line would be output as :

```
<rowData>
```

If the final parameter were also specified as FALSE, as follows :

```
OUTPUT (XMLROWDATA ('rowData', Row, TRUE);
```

the output would appear as :

```
<rowData>
  <Label>City</Label>
  <Value>Budapest</Value>
  <Code>37</Code>
</rowData>
```

### XmlFooter

The XMLFOOTER function returns an XML footer with the specified tag value.

```
StringVar := XMLFOOTER (HeaderTag);
```

## Manual Scope and Target Audience

### Parameters

HeaderTag : The value of the required closing tag

### Return values

A string containing a formatted XML footer.

### Remarks

The following line :

```
OUTPUT (XMLFOOTER ('items')) ;
```

is exactly equivalent to the line :

```
OUTPUT ('</items>') ;
```

## Text file functions

The text file functions provide an adequate set of functions to create or input text from text files. Functions are provided to read, write and perform simple formatting on text. Simple folder (path) manipulation is also provided.

### Createfile

The CREATEFILE function opens or creates a text file. The handle of the file is returned.

```
FileHandle := CREATEFILE (FileName);
```

#### Parameters

FileName	:	The name of the file to be opened or created. This should include the full path for the file.
bForce	:	Optional. A flag indicating whether the file should be opened regardless of existence of the nominated path.

#### Return values

The handle of the opened file.

#### Remarks

If the nominated file exists, it is opened and truncated (all content cleared). If the file does not exist, it is created as a blank file.

If bForce is specified as TRUE, a file will be created even if the nominated path is invalid. If the file fails to open / create on the nominated path, the system will first attempt to create the nominated folder. If this attempt fails as well, the file will be created in the system temporary folder. Provided that the system temporary folder is valid, a file will be created and the handle to this file will be returned.

The handle of the new or truncated file is returned. This handle may be used in subsequent instructions to write text output to the file.

### Writefile

The WRITEFILE function writes text information to an open file.

```
WRITEFILE (FileHandle, Value);
```

#### Parameters

FileHandle	:	A valid file handle, returned by a CREATEFILE or APPENDFILE call.
Value	:	The text to be written to the file

#### Return values

None.

#### Remarks

The nominated value may be of any simple type. If the value is text (a string) it will be output without formatting. If it is of a numeric or date type, it will be converted to a string first before output - the conversion rules are as for assigning a numeric, date or other value into a string.

Any text written to the file will result in the column index being incremented by the number of characters output. The LavaStream system will maintain the column index of the current output since the last linefeed. This is used when SPACEFILE is called to assert a required column spacing.

## Manual Scope and Target Audience

If carriage return or linefeed data is output directly to the file through WRITEFILE, the column index will be compromised - instead, use the WRITEFILENL function - this will correctly reset the column index so that subsequent use of the SPACEFILE function will result in the correct column.

### WritefileNL

The WRITEFILENL function writes a new line (carriage return / linefeed) character pair to an open file.

```
WRITEFILENL (FileHandle) ;
```

#### Parameters

FileHandle : A valid file handle, returned by a CREATEFILE or APPENDFILE call.

#### Return values

None.

#### Remarks

The WRITEFILENL function writes a pair of characters (0DX, 0AX) to the nominated file and resets the column index so that subsequent calls to SPACEFILE result in the correct column spacing.

### Readfile

The READFILE function reads text data from an open file, and moves the current file position to the end of the read segment.

```
StringVariable := READFILE (FileVar, TermString) ;
```

#### Parameters

FileHandle : A valid file handle, returned by an APPENDFILE call.  
TermString : A string fragment specifying the required termination criterion

#### Return values

A string containing the text read from the file up to but not including the position where the TermString value was first found

#### Remarks

The most conventional value specified for the TermString would be carriage return / linefeed (0DX & 0AX in LavaStream syntax) which will result in an entire text line being returned. However, any other valid string segment may be specified, such as a brace or semicolon if this is the appropriate termination for the line to be read.

The terminating fragment specified will be included in the string returned - for example, if the file contains (as the next text segment) the string "{abcde}..." and the TermString is specified as "{", the returned string segment will be "{abcde}".

If the end of the file is encountered before the TermString match, the remaining text segment is returned.

### Spacefile

The SPACEFILE function spaces the text output in the file to the nominated column.

```
SPACEFILE (FileHandle, ColumnValue) ;
```

#### Parameters

## Manual Scope and Target Audience

FileHandle : A valid file handle, returned by a CREATEFILE or APPENDFILE call.  
ColumnValue : The required column to which the output is to be spaced

### Return values

None.

### Remarks

The column count starts at column 1.

The operation is illustrated by the following example.

```
WRITEFILE (File1, 'abc');  
SPACEFILE (File1, 10);  
WRITEFILE (File1, 'xx');  
WRITEFILENL (File1);  
WRITEFILE (File1, 'testing');  
SPACEFILE (File1, 10);  
WRITEFILE (File1, 'xx');  
WRITEFILENL (File1);  
WRITEFILE (File1, 'offset');  
SPACEFILE (File1, 11);  
WRITEFILE (File1, 'zz');  
WRITEFILENL (File1);
```

The resultant output would be :

```
abc      xx  
testing  xx  
offset   zz
```

Note that in order for SPACEFILE to operate correctly, the function WRITEFILENL should be used to terminate lines - this notifies the file spacing algorithm that a new line has commenced.

## Closefile

The CLOSEFILE function closes a file previously opened using a CREATEFILE or APPENDFILE function.

```
CLOSEFILE (FileHandle);
```

### Parameters

FileHandle : The handle of the file to be closed.

### Return values

None.

### Remarks

If the file handle is invalid (does not represent an open file) the function has no effect.

## Appendfile

The APPENDFILE function opens the nominated file and sets the current file position to the end of the file.

```
FileHandle := APPENDFILE (FileName);
```

### Parameters

FileName : The name of the file to be opened.

### Return values

The handle of the opened file.

### Remarks

The nominated file must exist in the current default folder.

The APPENDFILE function should be used only to open text files. Results on non-text files could be unpredictable.

## Setfilepos

The SETFILEPOS function sets the current output offset to the nominated position.

```
SETFILEPOS (FileHandle, FilePos);
```

### Parameters

FileHandle	:	A valid file handle, returned by a CREATEFILE or APPENDFILE call.
FilePos	:	The required file position.

### Return values

None.

### Remarks

If the nominated file position is invalid (past the end of the file) the file position is not set.

If the file position is set successfully, subsequent WRITEFILE calls will output text to the nominated position.

## Setpath

The SETPATH function sets the current (default) folder.

```
CurrentPath := SETPATH(PathValue);
```

### Parameters

PathValue	:	The required folder (path)
-----------	---	----------------------------

### Return values

The previous default folder (path)

### Remarks

The nominated path should be valid - if the folder path cannot be found, the SETPATH call will have no effect.

## Generic file functions

The following functions operate on Windows files.

## FindFile

The FINDFILE function finds the first file matching a given (typically wildcarded) file specification

```
FINDFILE (FileSeek_id, FileSpec, FirstFile);
```

### Parameters

FileSeek_id	:	A returned seek identifier, must be used in subsequent calls to NextFile.
-------------	---	---

## Manual Scope and Target Audience

FileSpec	:	A string or string variable specifying a wildcarded string for the pattern filenames should match to be included in the file set
FirstFile	:	A string variable which receives the first matching filename

### Return values

None.

### Remarks

FindFile will return only files (not folders). To search for folders, use the FindFolder command.

In order to search a given folder, the path to the folder may either be included in the FileSpec parameter or the SETPATH command may be called prior to the FindFile call in order to establish the default folder.

Once FindFile has been successfully called, subsequent matches to the same file pattern may be obtained by repeated calls to NextFile.

## NextFile

The NEXTFILE function finds the next file matching a file seek initialized using the FindFile command.

```
NEXTFILE (FileSeek_id, NextFile);
```

### Parameters

FileSeek_id	:	A seek identifier, initialized through a call to FINDFILE.
NextFile	:	A string variable which receives the next matching filename

### Return values

None.

### Remarks

NextFile will return only files (not folders). To search for folders, use the FindFolder command.

The seek identifier must first be established through a call to FINDFILE.

Once the last file in the set has been returned, the final call to NextFile will return an empty string in the NextFile parameter, and the FileSeek\_id will be closed. Subsequent calls to NextFile will fail.

## FindFolder

The FINDFOLDER function finds the first folder matching a given (typically wildcarded) folder specification

```
FINDFOLDER (FolderSeek_id, FolderSpec, FirstFolder);
```

### Parameters

FolderSeek_id	:	A returned seek identifier, must be used in subsequent calls to NextFolder.
FolderSpec	:	A string or string variable specifying a wildcarded string for the pattern folder names should match to be included in the folder set
FirstFolder	:	A string variable which receives the first matching folder name

### Return values

None.

### Remarks

FindFolder will return only folders (not files). To search for files, use the FindFile command.

## Manual Scope and Target Audience

In order to search within a given folder, the path to the folder may either be included in the FolderSpec parameter or the SETPATH command may be called prior to the FindFolder call in order to establish the default folder.

Once FindFolder has been successfully called, subsequent matches to the same folder pattern may be obtained by repeated calls to NextFolder.

### NextFolder

The NEXTFOLDER function finds the next folder matching a folder seek initialized using the FindFolder command.

```
NEXTFOLDER (FolderSeek_id, NextFolder);
```

#### Parameters

FolderSeek_id	:	A seek identifier, initialized through a call to FINDFOLDER.
NextFolder	:	A string variable which receives the next matching folder name

#### Return values

None.

#### Remarks

NextFolder will return only folders (not files). To search for files, use the FindFile command.

The seek identifier must first be established through a call to FINDFOLDER.

Once the last folder in the set has been returned, the final call to NextFolder will return an empty string in the NextFolder parameter, and the FolderSeek\_id will be closed. Subsequent calls to NextFolder will fail.

### WinFileCopy

The WINFILECOPY function copies a Windows file from the nominated source to the nominated target folder.

```
WINFILECOPY (SourceFileName, TargetFileName);  
WINFILECOPY (SourceFileName, TargetFileName, Options);
```

#### Parameters

SourceFileName	:	The fully qualified path and name of the source file
TargetFileName	:	Either the desired folder or the full path and name of the target file
Options	:	An optional parameter controlling the copy process - see the remarks section

#### Return values

None

#### Remarks

If successful (depending on Windows permissions and existence of the source file) the nominated source file is copied to the nominated target (folder or file).

The path and name of both source and target files must be specified in Windows standard format, i.e. using backslash for folder separators. Long filenames and path names are supported. The maximum allowed length of a fully qualified path name is 260 characters.

If the Options parameter is specified, this allows validation of the target file - a value of 1 will abort the copy if the target file already exists. A value of 0 or omission of the parameter will copy the file to the target

unconditionally.

### WinFileDelete

The WINFILEDELETE function deletes the nominated Windows file

```
WINFILEDELETE (FileName) ;
```

#### Parameters

FileName : The fully qualified path and name of the file

#### Return values

None

#### Remarks

If successful (depending on Windows permissions and existence of the source file) the nominated file is deleted.

The path and name of the file must be specified in Windows standard format, i.e. using backslash for folder separators. Long filenames and path names are supported. The maximum allowed length of a fully qualified path name is 260 characters.

### WinFileMove

The WINFILEMOVE function moves a Windows file from the nominated source to the nominated target folder.

```
WINFILEMOVE (SourceFileName, TargetFileName) ;
WINFILEMOVE (SourceFileName, TargetFileName, Options) ;
```

#### Parameters

SourceFileName : The fully qualified path and name of the source file  
 TargetFileName : Either the desired folder or the full path and name of the target file  
 Options : An optional parameter controlling the move process - see the remarks section

#### Return values

None

#### Remarks

If successful (depending on Windows permissions and existence of the source file) the nominated source file is moved to the nominated target (folder or file). If successful, the source file will be removed.

The path and name of both source and target files must be specified in Windows standard format, i.e. using backslash for folder separators. Long filenames and path names are supported. The maximum allowed length of a fully qualified path name is 260 characters.

If the Options parameter is specified, this allows validation of the target file or options for the move process.

The following values are supported :

- |   |                  |   |
|---|------------------|---|
| 1 | Replace existing | If the target file exists, the function replaces its contents with the contents of the source file, provided that security requirements regarding access control lists are met. This value cannot be used if either SourceFileName or TargetFileName names a directory. |
| 2 | Copy allowed     | If the file is to be moved to a different volume, the function simulates the move by using the CopyFile and DeleteFile functions.   |
| 8 | Write through    | The function does not return until the file is actually moved on the disk. Setting this value guarantees that a move performed as a copy and delete operation is  |

flushed to disk before the function returns. The flush occurs at the end of the copy operation.

### WinFileRename

The WINFILERENAME function renames the nominated Windows source file to the target name specified.

```
WINFILERENAME (SourceFileName, TargetFileName);
```

#### Parameters

SourceFileName	:	The fully qualified path and name of the source file
TargetFileName	:	Either the desired folder or the full path and name of the target file

#### Return values

None

#### Remarks

If successful (depending on Windows permissions and existence of the source file) the nominated source file is renamed to the nominated target (folder or file). If successful and the target folder is not the same as the source folder for the file, the source file will be removed.

The path and name of both source and target files must be specified in Windows standard format, i.e. using backslash for folder separators. Long filenames and path names are supported. The maximum allowed length of a fully qualified path name is 260 characters.

### WinFileExists

The WINFILEEXISTS function tests for the existence of a nominated file on disk.

```
WINFILEEXISTS (FileName);
```

#### Parameters

FileName	:	The fully qualified path and name of a file to be checked.
----------	---	--

#### Return values

TRUE if the file exists, FALSE if not.

#### Remarks

The path and name of the file must be specified in Windows standard format, i.e. using backslash for folder separators. Long filenames and path names are supported. The maximum allowed length of a fully qualified path name is 260 characters.

### WinGetFolder

The WINGETFOLDER function obtains the current default folder (directory)

```
StringVar := WINGETFOLDER();
```

#### Parameters

None

#### Return values

The full path of the current default folder

#### Remarks

The folder (path) is returned Windows standard format, i.e. using backslash for folder separators.

### WinSetFolder

The WINSETFOLDER function sets the current default folder (directory)

```
WINSETFOLDER (FolderSpec) ;
```

#### Parameters

FolderSpec : The full path specification for the new default folder

#### Return values

None

#### Remarks

The folder (path) must be specified in Windows standard format, i.e. using backslash for folder separators.

### ReadMemFile

The READMEMFILE function reads a file from disk or other storage device and places the file content in a MEMFILE variable.

```
READMEMFILE (FileName, MemFileVar) ;
```

#### Parameters

FileName : The fully qualified path and filename of the required file  
MemFileVar : A MemFile variable which will store the content of the file

#### Return values

None.

#### Remarks

The FileName parameter must resolve the file required. If a SETPATH has been issued to the folder in which the file lies before the ReadMemFile is issued, the filename only may be used omitting the path.

Once the content of the file has been read into the MemFile variable, this content may be used to issue a SERVERPUTFILE command to write the file to the Lava Server.

Once the file has been fully processed, a FREEMEMFILE command should be issued to free the memory associated with the file content.

### FreeMemFile

The FREEMEMFILE command frees the memory for the file content in a MemFile variable.

```
FREEMEMFILE (MemFileVar) ;
```

#### Parameters

MemFileVar : A MemFile variable which contains the content of a file.

#### Return values

None.

#### Remarks

## Manual Scope and Target Audience

Once a FREEMEMFILE command has been issued on a MemFile variable, the content of the file stored in the variable is lost.

### ServerFetchFile

The SERVERFETCHFILE command fetches the content of a nominated file from the Lava Server and places it in the specified MemFile variable.

```
SERVERFETCHFILE (Folder, FileName, MemFileVar);
```

#### Parameters

Folder	:	The folder name under which the file was stored on the Lava Server.
FileName	:	The file name under which the file was stored on the Lava Server.
MemFileVar	:	A MemFile variable into which the content of the file will be placed.

#### Return values

None.

#### Remarks

A file may only be fetched from the Lava Server if a SERVERPUTFILE had previously been issued for the nominated file / folder combination.

Once a server file has been successfully fetched, the specified MemFile variable contains the entire content of the file in an allocated memory area.

In order to free the memory allocated when the file has been fully processed, the function FREEMEMFILE must be called.

### ServerPutFile

The SERVERPUTFILE command writes the content of a memory file to the Lava Server into a nominated folder and file.

```
SERVERPUTFILE (Folder, FileName, MemFileVar);
```

#### Parameters

Folder	:	The folder name under which the file is to be stored on the Lava Server.
FileName	:	The file name under which the file is to be stored on the Lava Server.
MemFileVar	:	A MemFile variable which contains the file content

#### Return values

None.

#### Remarks

In order to store a file using the SERVERPUTFILE function, the MemFile variable must have been used in either a READMEMFILE or SERVERFETCHFILE function, either of which would place the content of a file into the MemFile variable.

In order to free the memory allocated in the MemFile variable when the file has been fully processed, the function FREEMEMFILE must be called.

## WriteMemFile

The WRITEMEMFILE function writes a file to disk or other storage device from a MEMFILE variable which contains file content.

```
WRITEMEMFILE (FileName, MemFileVar);
```

### Parameters

FileName	:	The fully qualified path and filename of the required file
MemFileVar	:	A MemFile variable which contains the required file content

### Return values

None.

### Remarks

The FileName parameter must resolve the file required. If a SETPATH has been issued to the folder in which the file lies before the WriteMemFile is issued, the filename only may be used omitting the path.

Once the file has been fully processed, a FREEMEMFILE command should be issued to free the memory associated with the file content.

## Window Related Functions

### Message

The MESSAGE function displays a message box according to parameters provided. Only functions in a Windows application.

```
Selection := MESSAGE(Caption, Message, Button1, Button2, Button3, Button4);
```

#### Parameters

Caption	:	The required caption for the message box
Message	:	The message to be displayed in the message box
Button1	:	The text to appear on the first button
Button2	:	(Optional) The text to appear on the second button
Button3	:	(Optional) The text to appear on the third button
Button4	:	(Optional) The text to appear on the fourth button

#### Return values

The ordinal (1-based) of the button clicked by the user. For example, if the user clicks on the first button, the return value will be 1, on the second button it will be 2 and so on. The maximum value returned is limited to the number of buttons specified, or 4 if all buttons are specified.

#### Remarks

The text (caption) for the first button must be specified, as a message box must have at least one response button. Subsequent buttons are optional, and will be omitted if the corresponding parameter is omitted.

The buttons are sized to fit the text specified, to a practical maximum allowed by the message box dimensions of around 50 characters per button.

For an example of a code segment using a message box, see the example Message Box in the examples chapter.

## Miscellaneous functions

### Exec

The EXEC function allows execution (spawning) of a new process.

```
EXEC (ProcessName, Parameters);
EXEC (ProcessName, Parameters, BatchExec);
EXEC (ProcessName, Parameters, BatchExec, WaitValue);
EXEC (ProcessName, Parameters, BatchExec, WaitValue, ServerExec);
```

#### Parameters

ProcessName	:	The path and name of an executable or batch file
Parameters	:	A string value or variable specifying the parameters to be passed to the new process. This may be an empty string.
BatchExec	:	An optional parameter specifying that the command to be executed is a batch file (.bat command) - this must be set to TRUE for batch files to execute successfully. The default is FALSE.
WaitValue	:	An optional parameter specifying whether the LavaStream engine should wait for the process to complete before continuing execution. The default is FALSE.
ServerExec	:	An optional parameter specifying whether the nominated command is executed on the server, instead of on the client. The default is FALSE.

#### Return values

None.

#### Remarks

If the specified file cannot be found or is not a valid executable or batch file, processing will fail.

If LavaStream execution should be suspended until completion of the new process, specify TRUE in the final (optional) parameter. The default is FALSE, i.e. execution continues immediately the new process has been started. Note that specifying TRUE here requires the new process to complete and close - while any part of this process is still executing, LavaStream execution will be suspended.

If execution is specified on the server, batch file execution is not supported.

### WinFormatMessage

The WINFORMATMESSAGE function formats a nominated Windows error (as determined by WinGetLastError) into a descriptive string.

```
StringVar := WINFFORMATMESSAGE ();
StringVar := WINFFORMATMESSAGE (ErrorCode);
```

#### Parameters

ErrorCode	:	An optional parameter specifying a Windows error code, determined using GetLastError
-----------	---	--

#### Return values

A string containing the description of the error code as returned by the Windows operating system.

### WinGetLastError

## Manual Scope and Target Audience

The WINGETLASTERROR function determines the last error code resulting from an operating system call

```
IntegerVar := WINFGETLASTERROR();
```

### Parameters

None

### Return values

An integer containing the last Windows error code

### Remarks

In order to format the returned error code into a descriptive string, you may use WINFORMATMESSAGE.

## CreateNode

The CreateNode function allows a new vertex node to be created

```
Node_id := CREATENODE(OwnerNode_id, Label, Path, NodeType, x, y);
```

### Parameters

OwnerNode_id	:	The Row ID for the parent vertex node
Label	:	A string specifying the label for the new node
Path	:	A string specifying the full path (including filename) for the new node
NodeType	:	An integer value specifying the node type for the node
x, y	:	Blueprint workspace co-ordinates for the node

### Return values

The ID for the newly created node. If the creation fails, the ID returned is 0.

### Remarks

The path and name of the file for the node (Path parameter) are specified in Windows standard format, i.e. using backslash for folder separators. Long filenames and path names are supported. The maximum allowed length of a fully qualified path name is 260 characters.

The owner node must be a valid, existing vertex node, and should be of an appropriate type for the new subordinate node specified.

## NodePath

The NodePath function determines the file path for a nominated vertex node (if the attribute has been configured)

```
StringPath := NODEPATH(Node_id);
```

### Parameters

Node_id	:	The Row ID of a vertex node for which the filepath is required
---------	---	--

### Return values

The fully qualified path of the node. This includes the complete windows path and the filename and extension.

### Remarks

The path and name of the file are specified in Windows standard format, i.e. using backslash for folder separators. Long filenames and path names are supported. The maximum allowed length of a fully qualified path name is 260 characters.

### ResolvePath

The ResolvePath function determines the file path for the nominated path ID

```
StringPath := RESOLVEPATH(Path_id, Filename);
```

#### Parameters

Path_id	:	A valid Blueprint path ID, fully specified for the current user
Filename	:	An optional filename to be appended to the path once resolved

#### Return values

The fully qualified path for the path ID, as resolved for the current user. If specified, the nominated filename is appended to the folder resolved by the path ID. The resulting path is a complete windows path

#### Remarks

The path and name of the file are specified in Windows standard format, i.e. using backslash for folder separators. Long filenames and path names are supported. The maximum allowed length of a fully qualified path name is 260 characters.

### Sleep

The SLEEP function implements a delay based on the Windows Sleep function.

```
SLEEP(TimeValue);
```

#### Parameters

TimeValue	:	The time to sleep, in milliseconds
-----------	---	------------------------------------

#### Return values

None.

#### Remarks

The LavaStream engine will sleep for at least the nominated number of milliseconds.

For this period, no processing (cpu activity) will occur.

The Sleep function is quite accurate, and in general terms the delay will be at least the period specified and normally not more than 1 or perhaps 2 milliseconds longer - this delta, though, will depend on the current processing load and the speed of the CPU.

### SendMail

The SENDMAIL function sends an electronic mail message through a nominated SMTP server..

```
SENDMAIL(MailServer, From, To, Subject, Message, Attachments);
```

#### Parameters

MailServer	:	A valid SMTP mail server, specified either as a web address or as an IP address
From	:	The mail address of the sender
To	:	The mail address of the recipient
Subject	:	The subject of the mail message
Message	:	The body (message) of the mail
Attachments	:	(Optional) semicolon-separated list of file attachments

### Return values

None. An error code may be obtained through the LAVASTATUS function.

### Remarks

The MailServer nominated must be a valid, reachable SMTP mail server. If specified as a web name address, the typical form would be "mail.servername.com". If the address is invalid or unreachable at the time of transmission, the mail will fail and set an error code.

The recipient address is a conventional e-mail address, formed in the usual way - for example ["fred@bloggs.com"](mailto:fred@bloggs.com).

The message may be any text, including HTML if the receiving mail client supports hypertext mail.

Attachments may optionally be specified in the final parameter. If specified, the full path of each attachment should be specified, with multiple attachments separated by a semicolon ( ; ). The final attachment specified should not be followed by a semicolon.

Note that the SMTP protocol does not require passwords. However, access to the mail server may be limited by firewalls - the originating workstation or server (on which the LavaStream code is being executed) must have access to the mail server through the SMTP port.

## SetDecimals

The SETDECIMALS function sets the default number of decimals to be used for Float output for the current LavaStream session

```
SETDECIMALS (DecimalCount) ;
```

### Parameters

DecimalCount : The number of decimals to display

### Return values

None.

### Remarks

String output of Float values will display the specified number of decimals after this call.

The value is only valid for the current LavaStream session - on commencing a new session, the output decimals will once again default to 2.

## Thread

The THREAD function starts a LavaStream procedure in a Windows thread

```
THREAD (LavaCommandString) ;
```

### Parameters

LavaCommandString : A correctly formed LavaStream command string, in the form STREAM.Project.Module.Procedure(parameters).

### Return values

None.

### Remarks

## Manual Scope and Target Audience

A new thread is started and the nominated Lava procedure is executed in the thread. A new WebSession is created for the procedure. When the procedure completes, the thread is automatically closed.

When running under Apache, a practical limit of 120 concurrent threads is

### WriteLog

The WRITELOG function creates an entry in the Apache LavaStream log file

```
WRITELOG (String) ;
```

#### Parameters

String : Any constructed or defined string

#### Return values

None.

#### Remarks

This function will only function under Apache. When executing a LavaStream procedure from the Stream Elements environment or from the Blueprint Editor, the function will have no effect.

When executing LavaStream under Apache, logging must be enabled (Logging=true in the QubikServer.ini file). A log entry will be produced regardless of the logging level (LogLevel) setting.

For details on configuration of Apache and available settings in the QubikServer.ini file, see the section [Apache Configuration and Testing](#) in this manual.

# Apache Configuration and Testing

The LavaStream language has three major purposes.

The first is as a stored function language within Lava SQL, to provide a language extension to SQL.

The second is as a scripting language to allow for manipulation of a Lava database, or importing data from ODBC targets.

The third is to allow simple through complex procedures and programs to be written for execution under Apache Web Server, triggered from browser applications. This third purpose for LavaStream is discussed in this chapter.

When LavaStream is used under Apache, it serves a purpose similar to that of PHP, allowing programs to be designed and written which interface to a database (Lava, an ODBC database or both) in order to provide complex, data related functionality to a browser application.

## Installing Apache

If you already have Apache installed on your intended Web Server, ensure that the release is 2.2 or later (2.2.4 or later preferred). If not, please uninstall your current version and install 2.2.4 as described below.

### Installing under Windows XP or Windows Server 2003 (Windows 5.1 / 5.2)

Installing Apache 2.2.4 under Windows 5.1 or 5.2 should be no problem whatsoever. Ensure that you have administrative privileges for the selected server, and simply execute the Apache install. You should not encounter any problems, and you should have Apache running within a few minutes.

To check that Apache is working correctly after installation, check the website <http://localhost> from any browser invoked from the Apache server. It should display "It works!". If not, invoke the Apache configuration test interface (Start - All Programs - Apache HTTP Server 2.2.x - Configure Apache Server - Test Configuration). Follow the instructions to correct any problems with the configuration.

### Installing under Windows Vista

To install Apache under Windows Vista, some undocumented footwork is required. Please follow the following steps accurately.

1. Uninstall any previous installations of Apache Web Server.
2. Ensure that all folders for previous installations of Apache are completely removed - manually delete them if any persist.
3. Turn off the firewall.
4. Stop User Account Control. (Start - Settings - Control Panel - User accounts; Select "Turn user account control on or off", unflag UAC, select OK.)
5. Download the most recent version of Apache from <http://httpd.apache.org/download.cgi> and place it on the desktop. Rename the file to *apache.msi*.
6. Ensure that you are logged on to the workstation with the Administrator account.
7. From the Start button, select Start - All Programs - Accessories, right-click on Command Prompt and choose the option "Run as Administrator"
8. Change the default directory to the desktop folder (cd desktop)
9. Start the install using the command "msiexec /i apache.msi"

## Manual Scope and Target Audience

10. Select installation options as required, with the preferred installation folder being c:\apache. Other desired options may work, or not; if they do not, repeat and choose c:\apache. This will work.
11. Reboot the workstation.
12. The installation will place the startup command for the Apache monitor into the Start - All Programs - Startup folder; this will not work under Vista. Remove it manually.
13. From a browser, check the website <http://localhost>. This should display "It works!". If not, invoke the Apache configuration test interface (Start - All Programs - Apache HTTP Server xxx - Configure Apache Server - Test Configuration). Follow the instructions.
14. Turn the firewall back on. Ensure that port 80 is permitted as an exception. Apache should now be functional.

## Apache Configuration

In order to enable LavaStream functionality under Apache, the following is required :

1. A working installation of Apache 2.2 or later.
2. The Apache installation must be on a Windows server (or workstation) running Windows 5.1 or later (this equates to Windows XP, Windows Server 2003 or Windows Vista at the time of writing).
3. The Lava Apache Module must be installed and configured for Apache as described below
4. The Lava ODBC driver must be correctly installed on the Apache Server or Workstation.

### Using the Qubik Apache Install

The recommended process for installing LavaStream support under Apache is to run the Qubik Apache Install utility (Qubik Apache Install.exe). This installation utility will do all that needs to be done in order to link LavaStream to Apache. If you wish to verify the installation, or you want to do the installation manually in order to modify some of the parameters, follow the steps listed in the section **Manual Apache Configuration**, below.

Note that the Qubik Apache install does not install the Qubik Lava ODBC driver - this must be done separately.

## Manual Apache Configuration

The Apache configuration process is automated in the utility Qubik Apache Install.exe, as described above. If you wish to verify or perform this installation manually, follow the steps listed below.

### Apache Configuration File

To configure the Lava Apache Module, adjustments need to be made to the Apache configuration file.

The configuration file may either be opened Start menu (Start - All Programs - Apache HTTP Server xxx - Configure Apache Server - Edit the Apache httpd.conf Configuration File) or be invoking a text editor on the file directly (it is located under the Apache installation folder in the folder conf, and is called httpd.conf).

Two amendments need to be made. The first is around line 115 of the default configuration file, just below the section which should end as follows :

```
#LoadModule unique_id_module modules/mod_unique_id.so
LoadModule userdir_module modules/mod_userdir.so
#LoadModule usertrack_module modules/mod_usertrack.so
#LoadModule vhost_alias_module modules/mod_vhost_alias.so
#LoadModule ssl_module modules/mod_ssl.so
```

Right below this section, add the line :

```
LoadModule lava_module modules/mod_lava.so
```

The final amendment may be made right at the end of the file. Add the following section :

```
# DBX
<Location /dbx>
  AuthType LAVA
  Require valid-user
  DirectoryIndex main.html
  ErrorDocument 403 /dbx/forbid.html
  LAVA:ScriptMatch +|*.xml
</Location>
```

If, for whatever reason, you already have a dbx folder below the default Apache document folder, or you do not wish to use this folder name, you may change this but be sure to consistently change every reference in this configuration to your selected folder name.

### Lava Apache Module

Copy the Lava Apache Module (**mod\_lava.so**) to the **modules** folder, under the main Apache installation folder.

### Lava ODBC Driver

Install the Lava ODBC Driver on the Apache Server or Workstation. Ensure that the ODBC Driver release is the same as the release of your Lava Server. The installation file should be **QubikODBCSetup.exe**.

### Lava Javascript Library

Finally, the Lava Javascript Library must be copied to the Apache document folder. The default document folder is called **htdocs** and is found under the main Apache installation folder. Copy the Lava **dbx** folder (which contains the Javascript library) into the Apache document folder. If you have changed the default name of the dbx extension in the httpd.conf file above, ensure that this matches the name of the folder placed in the Apache document folder.

## Setting up the Apache Lava Client

The Lava Module now configured to run under Apache needs to know where to find your Lava Server. This must be configured in the QubikServer.ini file, which will be located (or, if it does not exist, must be created) in your Windows folder (c:\windows in standard installations). The contents of this file should be as follows :

```
[Logon]
Logging=false
LogLevel=0
Server=LavaServer
ClientPath=x:\temp
User=;ê@:*5ö
Password="yF}/*5
```

All currently supported parameters must be in the [Logon] section.

The parameters provided for in the .ini file are as follows :

### Logging

The Logging variable specifies whether the Apache Lava Module will create and write to a log file to allow tracing and diagnostics. To turn on logging, the following entry

```
Logging=true
```

must be placed in the **Logon** section of the ini file.

If logging is enabled, a log file by the name LavaStream.log is placed in the same folder specified in the ClientPath parameter, described below.

### LogLevel

Different levels of logging are supported. The most basic level provides the minimum of logging and will keep log entries down to the absolutely essential information.

```
LogLevel=1
```

At level 2, more information is provided at the cost of more rapid growth of the log file.

```
LogLevel=2
```

At level 3, verbose information including benchmark timing and detailed output information (as sent through Apache to the browser) is reported.

Regardless of the level of logging selected, if logging is enabled the WRITELOG function will result in visible output to the log file.

### Server

The Server parameter nominates the Lava server the Apache client will use. This will normally be specified as a local machine name, but may be specified as an IP address or a URL.

```
Server=LavaServer
```

```
Server=192.168.1.155
```

```
Server=www.qubik.net
```

### ClientPath

The ClientPath parameter specifies a valid, existing folder used for temporary Lava client data

```
ClientPath=c:\temp
```

At least 50 Mb of free space should exist on the drive nominated.

### User

The User parameter specifies the username to enable the Apache Lava Module to log into the Lava Server. Normally, this will be the system user (unless the database administrator has reasons to explicitly limit access

from the Apache client).

However, the User parameter cannot be set using readable text for security reasons (Lava users and passwords are not specified in text files as this may allow for security breaches).

An example of a User specification could be :

```
User=5šİ€0o<
```

The User and Password parameters are set using the ServerLogon utility, accessible from the Start menu.

### Password

The Password parameter specifies the Lava Server password parameter used by the Apache Lava Module to log into the Lava Server. This must match the correct password for the specified user.

The Password parameter cannot be set using readable text for security reasons (Lava users and passwords are not specified in text files as this may allow for security breaches).

An example of a Password specification could be :

```
Password=óbn`yF}
```

The User and Password parameters are set using the ServerLogon utility, accessible from the Start menu,

## Testing with Apache

In order to provide a means of testing functionality while running under Apache, a logging facility is provided. Using the WRITELOG command in LavaStream, output may be directed to the Apache LavaStream log file. In order to enable this feature, logging must be enabled in the QubikServer.ini file, as follows :

```
Logging=true
```

Provided that logging is enabled, output may be directed to the log file as in the following example :

```
WRITELOG(`Counter : ` & Counter);
```

The log file will be located in the folder specified for the ClientPath in the QubikServer.ini file, and is named LavaStream.log.

Although it is possible to debug LavaStream procedures using the WRITELOG function, this is not the primary purpose. Normally, only crucial information such as parameters originating from the browser are written to the log, after which the LavaStream debugger is used to do further diagnostic work.

## FormatX - Styles and Attributes

A format styles module is supplied which provides the constants required by Lava Functions.

The first set is for the FORMAT command as primary and secondary formats. See the FORMAT command entry in the text output function section above for use of this function.

The second set specifies attributes used in the CREATETABLE command.

```

MODULE FormatX;
CONST
  FORMAT_P_NUMBER          - = 001H;
  FORMAT_P_CURRENCY        - = 002H;
  FORMAT_P_ACCOUNTING      - = 003H;
  FORMAT_P_DATE            - = 004H;
  FORMAT_P_TIME            - = 005H;
  FORMAT_P_PERCENTAGE      - = 006H;
  FORMAT_P_SCIENTIFIC      - = 007H;

  FORMAT_P_HEXQUAD        - = 008H;
  FORMAT_P_VDT            - = 009H;
  FORMAT_P_IP             - = 00AH;

  FORMAT_P_BITSET         - = 00BH;
  FORMAT_P_BOOLEAN        - = 00CH;

  FORMAT_P_KB_QUAD        - = 010H;          (* Kilobyte (derived) *)
  FORMAT_P_MB_QUAD        - = 011H;          (* Megabyte (derived) *)

(* Number Formats - Secondary *)
  FORMAT_S_NULL           - = 0;

  FORMAT_S_BASE_BINARY    - = 1;
  FORMAT_S_BASE_HEX       - = 2;

  FORMAT_S_NEGATIVE_1     - = 0;          (* -1,234.10      *)
  FORMAT_S_NEGATIVE_3     - = 2;          (* (1,234.10)    *)

  FORMAT_S_DATE_1         - = 1;          (* dd/mm/yy      *)
  FORMAT_S_DATE_2         - = 2;          (* dd/mm/yyyy    *)
  FORMAT_S_DATE_3         - = 3;          (* mm/dd/yy      *)
  FORMAT_S_DATE_4         - = 4;          (* mm/dd/yyyy    *)
  FORMAT_S_DATE_5         - = 5;          (* mmm dd, yyyy  *)
  FORMAT_S_DATE_6         - = 6;          (* mmmmmmmmm dd, yyyy *)
  FORMAT_S_DATE_7         - = 7;          (* dd mmm yy     *)
  FORMAT_S_DATE_8         - = 8;          (* dd mmm yyyy   *)
  FORMAT_S_DATE_9         - = 9;          (* yyyy/mm/dd    *)

  FORMAT_S_TIME_1         - = 1;          (* hh:mm         *)
  FORMAT_S_TIME_2         - = 2;          (* hh:mm:ss     *)
  FORMAT_S_TIME_3         - = 3;          (* hh:mmXM      *)
  FORMAT_S_TIME_4         - = 4;          (* hh:mm:ssXM   *)
  FORMAT_S_TIME_5         - = 5;          (* hhmm         *)
  FORMAT_S_TIME_6         - = 6;          (* hhmmss       *)

  SQL_OBJECT_TABLE       - = 1H;          (* Object is a conventional table *)

```

## Manual Scope and Target Audience

```
SQL_OBJECT_VIRTUAL      - = 00010000H; (* Table is virtual (non-physical) *)
SQL_OBJECT_RAW          - = 00020000H; (* Table has no version group *)
SQL_OBJECT_PERSISTENT  - = 00040000H; (* Flags virtual table as persistent *)
SQL_OBJECT_FRAMED      - = 00100000H; (* Table supports transaction frames *)
SQL_OBJECT_LOCAL        - = 0001H;  (* Table is created in the client database *)
SQL_OBJECT_SERVER      - = 0002H;  (* Table is created on the server *)
```

```
END FormatX.
```

# LavaStream Examples

## Coding Techniques

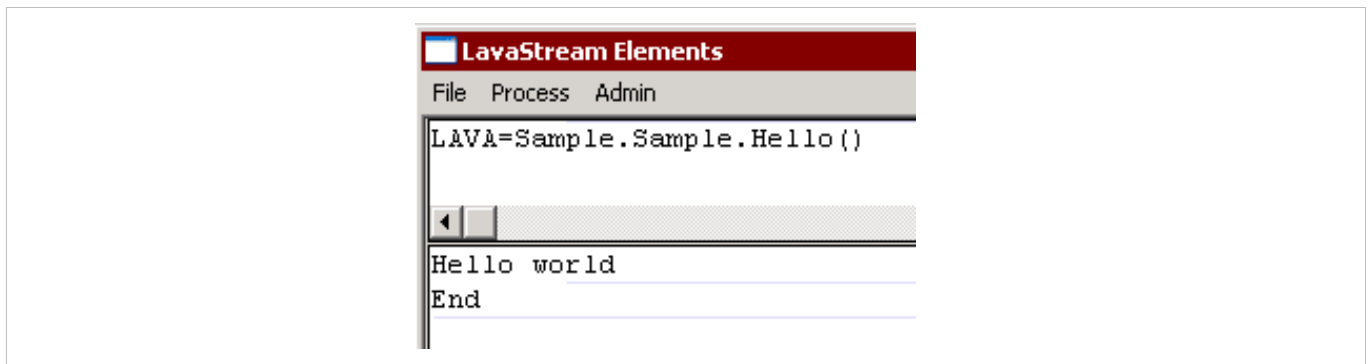
### Very Simple Program

The following simple test sample illustrates some basic coding principles in LavaStream. The program is intended to be run in the LavaStream Elements test environment (or, of course, in the Blueprint Editor) and illustrates basic test output.

```
CONST
  NEWLINE      =  0DX & 0AX;

PROCEDURE Hello();
VAR
BEGIN
  OUTPUT('Hello world' & NEWLINE);
  OUTPUT('End');
END Hello;
```

If the above code is entered into a module by the name of **Sample** belonging to a LavaStream project by the name of **Sample**, the following LavaStream Elements image capture would exactly mimic the results of running the code.



### Temporary Table as Array Replacement

This example illustrates the usage of a Lava table to provide an array definition for a LavaStream program. The row type definition may be as simple or as complex as required, and there is virtually no limit to the number of elements that such a table will allow access to (although limited by workstation memory, typically access to tens of millions of array elements should not be a problem). As the most common problem with arrays is the correct definition of array size, this method effectively eliminates the problem.

```
(* Define the table structure *)

TYPE
  ResultRow      =  ROWTYPE
    user_name    :  STRING;
    schema_name  :  STRING;
  END;

PROCEDURE TableExample();
VAR
```

## Manual Scope and Target Audience

```
TempTable      : TABLE;
TempRow        : ResultRow;

BEGIN

(* Create a temporary table *)

TempTable := CREATETABLE('TempTable', ResultRow);

CLEAR(TempRow);
TempRow.user_name := 'Fred';
TempRow.schema_name := 'Design';
(*.Putting to a row beyond the end of the new table creates the row automatically *)
TempTable[10] := TempRow;
(*.Delete a row *)
CLEAR(TempTable[5]);
(*.Access a row - just as for array access *)
TempRow := TempTable[7];
(*.Drop the temporary table to free allocated memory *)
DROPTABLE(TempTable);
END TableExample;
```

Note that in the above example the final instruction drops the table. Although a temporary table as in the above example will automatically be dropped by the database kernel when the session is closed, if the DropTable is not executed the table will persist until the session is closed. In some cases this will not matter, but in some cases this will cause problems. For example, if the above example is run through the debugger, the first attempt will work correctly, but the second attempt will fail. The reason for this is that if the table is not dropped, it will persist to the second execution (unless the editor is closed between attempts). This will cause the CreateTable to fail on the second execution as it is not permissible to create two tables by the same name in the same schema. All the subsequent code will fail as a result of this.

In general, it is good practice to test the creation of the table, as follows :

```
TempTable := CREATETABLE('TempTable', ResultRow);
IF LAVASTATUS() # 0 THEN
(* do something because the create failed *)
END;
```

## Windows Programming

### Message Box

The following simple example illustrates the use of the MESSAGE command in a Windows program.

## Lava Database Access

### Simple Loop on Lava Table

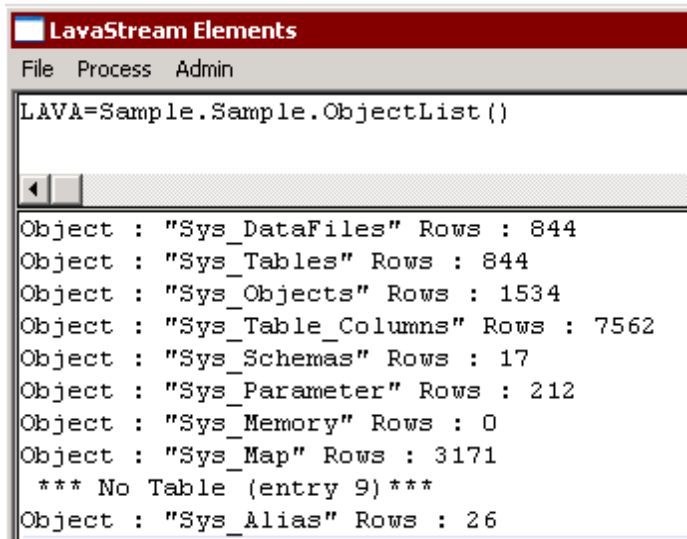
The following code segment illustrates the use of array access to a Lava system table, and access to the columns of the table.

```

PROCEDURE ObjectList();
VAR
(*.Define row variable for System_ControlFile table *)
  ObjectRow    : LAVA.SYSTEM.System_ControlFile.TYPE;
  Object_id    : INTEGER;
  rc           : INTEGER;
BEGIN
(*.Loop through first 10 tables *)
  FOR Object_id := 1 TO 10 DO
    ObjectRow := LAVA.SYSTEM.System_ControlFile[Object_id];
(*....Test for valid entry *)
    IF LAVASTATUS() # 0 THEN
      OUTPUT(' *** No Table (entry ' & Object_id & ')***' & NEWLINE);
      CYCLE;
    END;
(*....Output sample data from controlfile entry *)
    OUTPUT('Object : "' & ObjectRow.ObjectName & '" Rows : ' & ObjectRow.UsedRows
& NEWLINE);
  END;
END ObjectList;

```

If run in LavaStream Elements, the following output should be generated :



```

LAVA=Sample.Sample.ObjectList()
Object : "Sys_DataFiles" Rows : 844
Object : "Sys_Tables" Rows : 844
Object : "Sys_Objects" Rows : 1534
Object : "Sys_Table_Columns" Rows : 7562
Object : "Sys_Schemas" Rows : 17
Object : "Sys_Parameter" Rows : 212
Object : "Sys_Memory" Rows : 0
Object : "Sys_Map" Rows : 3171
*** No Table (entry 9)***
Object : "Sys_Alias" Rows : 26

```

*LavaStream Elements Interface*

## ODBC Database Access

Access to ODBC source databases is quite extensive, and in addition to the functionality as described below the **Blueprint** environment includes the following :

- Automatic retrieval of table definitions from an ODBC source into the **Blueprint** dictionary
- Automatic generation of SQL scripting to define equivalent tables in a Lava database
- Automatic generation of LavaStream source code for import of data from the ODBC source into the Lava table

For further information on this functionality, see the *Blueprint Operation Guide*.

### Retrieving ODBC Data

In order to provide a meaningful example of ODBC import techniques, the following is a more lengthy example illustrating a number of methods which may be used with ODBC data sources. In this example, the source is a Sage 500 database resident on Microsoft SQL Server. The code as provided is working and tested, and will allow import of the invoice header information from an actual Sage database.

The existence of a Lava data table with appropriate format to receive the invoice header information is assumed. The required table could be created using the following SQL script segment (which assumes the existence of an ACCOUNT schema) :

```
CREATE [SERVER] TABLE Account.invoice_header (
  Version          VERSION,
  cust_no          STRING(10),
  branch_no        INTEGER,
  inv_no           STRING(10),
  order_no         STRING(10),
  inv_date         DATE,
  ord_date         DATE,
  del_date         DATE,
  status           STRING(2),
  rep_code         STRING(5),
  exp_date         DATE,
  addr_no          STRING(10),
  region_no        STRING(5),
  period           STRING(2),
  fyear            STRING(4)
)
  REPLICATE
  RECLAIM
  FRAMED;
```

In addition, the following row type must be defined to accept the ODBC data :

## Manual Scope and Target Audience

```
SageInvHeadType      = ROWTYPE
(* Note the use of QUAD for MS SQL dates *)
  customer           : STRING[100];
  ord_date           : QUAD;
  exp_date           : QUAD;
  del_date           : QUAD;
  addr_no            : STRING[100];
  region             : STRING[100];
  order_no           : STRING[100];
  status             : STRING[100];
  rep_code           : STRING[100];
  inv_no             : STRING[100];
  inv_date           : QUAD;
  period             : STRING[100];
  fyyear             : STRING[100];
END;
```

The code below will import the invoice header data :

```
PROCEDURE InvoiceHeadImport() : INTEGER;
VAR
  Connect           : INTEGER;
  InvHeadRecord     : LAVA.Account.invoice_header.TYPE;
  SageInvHeadRecord : SageInvHeadType;
  SageInvHead       : TABLE;
  InvHead_id        : INTEGER;
  RowCount          : INTEGER;
  Index             : INTEGER;
  rc                : INTEGER;
BEGIN
(*.Import data from MS SQL *)
  Connect := ODBC_CONNECT(
    'DRIVER=SQL Server;' &
    'SERVER=WindowsSQLserver;' & (* use correct server name *)
    'UID=username;' & (* use correct user name *)
    'PWD=password;' & (* use correct password *)
    'WSID=WorstationName;' & (* use correct workstation name *)
    'Network=DBMSSOCN');

  SageInvHead := ODBC_SQL(Connect,
    "select " &
    "invoice_customer, date_entered, date_required, date_despatched, customer, " &
    " region,order_no, status, class, invoice_no, " &
    " invoice_date, period, slyear " &
    "from " &
    "FAT.scheme.opheadm ");
    "where " &
    "status = '8' and slyear >= '2007'");
  ODBC_CLOSE(Connect);

(*.Assert Autocommit. No need for rollback on import data *)
  AUTOCOMMIT();
  SageInvHead := InvHeadImport();

  RowCount := TABLEROWS(SageInvHead)
  OUTPUT('Rows returned : ' & nCount & NEWLINE);
(*.Reserve adequate rows to avoid repeated row reserves during import *)
  rc := RESERVEROWS(LAVA.Account.invoice_header, RowCount);
```

## Manual Scope and Target Audience

```
FOR Index:= 1 TO RowCount DO
(*....Read data from ODBC import table *)
  SageInvHeadRecord := SageInvHead[Index];
  rc := LAVASTATUS();
  IF rc # 0 THEN
    OUTPUT('invalid invoice header row : ' & Index);
    OUTPUT(NEWLINE);
    CYCLE;
  END;
(*....Translate data to Lava table format *)
  CLEAR (InvHeadRecord);
  InvHeadRecord.cust_no      := SageInvHeadRecord.customer;
  InvHeadRecord.ord_date    := MSDATE(SageInvHeadRecord.ord_date);
  InvHeadRecord.del_date    := MSDATE(SageInvHeadRecord.del_date);
  IF MSDATE(SageInvHeadRecord.exp_date) < 0 THEN
    InvHeadRecord.exp_date  := 0
  ELSE
    InvHeadRecord.exp_date  := MSDATE(SageInvHeadRecord.exp_date)
  END;
  InvHeadRecord.addr_no     := SageInvHeadRecord.addr_no;
  InvHeadRecord.region_no  := SageInvHeadRecord.region;
  InvHeadRecord.order_no   := SageInvHeadRecord.order_no;
  InvHeadRecord.status     := SageInvHeadRecord.status;
  InvHeadRecord.rep_code   := SageInvHeadRecord.rep_code;
  InvHeadRecord.inv_no     := SageInvHeadRecord.inv_no;
  InvHeadRecord.inv_date   := MSDATE(SageInvHeadRecord.inv_date);
  InvHeadRecord.period     := SageInvHeadRecord.period;
  InvHeadRecord.fyear      := SageInvHeadRecord.fyear;
  InvHeadRecord.branch_no  := PRESSINGS;
(*....Acquire a row ID for the newly imported row *)
  InvHead_id := FREEROW(LAVA.Account.invoice_header);
(*....Write translated row to Lava table *)
  LAVA.Account.Invoice_header[InvHead_id] := InvHeadRecord;
  rc := LAVASTATUS();
  IF rc # 0 THEN
    OUTPUT('invoice row add failed : ' & InvHead_id &
          ' : invoice : ' & InvHeadRecord.inv_no );
    OUTPUT(NEWLINE);
    CYCLE;
  END;
END;
OUTPUT('Lava Invoice Rows : ' & TABLEROWS(LAVA.Account.invoice_header) &
      NEWLINE);
END InvoiceHeadImport;
```

## **Interfacing to Excel**

Excel spreadsheets may be accessed directly using the ODBC interface.

```
SELECT * FROM [Sheet1$];
```

```
INSERT INTO [Sheet1$] (Name, Phone, Job) VALUES (?, ?, ?);
```

## XML Interfacing

### Simple XML Output

This example shows how to output basic XML for a row type.

```

TYPE
  FormType          =  ROWTYPE
  customer          :  STRING[100];
  address           :  STRING[100];
  country           :  STRING[100];
END;

PROCEDURE SimpleXML();
VAR
  Form              :  FormType;
BEGIN
  Form.customer := 'John Smith';
  Form.address := '238 Stream Street, London';
  Form.country := 'England';
  Form.ROWID := 234; (* optional *)
  OUTPUT(XMLHEADER(TRUE, 'items'));
  OUTPUT(NEWLINE);
  OUTPUT(XMLROWDATA('formdata', Form));
  OUTPUT(NEWLINE);
  OUTPUT(XMLFOOTER('items'));
END SimpleXML;

```

Note that the NEWLINE outputs between the XML outputs are redundant, and are merely included to separate the output into lines of text. Normally these would be eliminated.

The following XML will be produced :

```

<?xml version="1.0" encoding="utf-8" ?> <items>
  <formdata ROWID="234">    <customer>John Smith</customer>    <address>238 Stream Street,
London</address>    <country>England</country> </formdata>
</items>

```

Note the ROWID attribute in the above XML - this is produced to allow interface to client forms, so that the database row information is preserved. If the ROWID attribute to the form is zero, this information is omitted.

### Basic XML input

In the following example, it is assumed that the XML as specified below is provided in the form of a string parameter to the **XMLinput** procedure. In addition, the assumption is that the schema **Test** exists and contains the table **FormList**, which itself contains the columns session, name and title.

```

<?xml version="1.0" encoding="utf-8" ?>
<items>
  <formdata ROWID="17">
    <session>327</session>
    <name> John</name>
    <title>mr</title>
  </formdata>
</items>

```

## Manual Scope and Target Audience

The input parameter in the following procedure would probably stem from Apache, having originally been created in a browser-based form containing the fields listed.

```
PROCEDURE XMLinput(pFormXML : STRING);
VAR
  Form          : LAVA.Test.FormList.TYPE;
BEGIN
  Form := pFormXML;
  LAVA.Test.FormList[Form.ROWID] := Form;
  COMMIT();
END XMLinput;
```

Note the use of the ROWID attribute to determine the data row to be updated. This is conventional practise in LavaStream, and this ROWID would be communicated to the browser from via XML output, to be echoed back when the form is accepted.

### XML List Output

This example shows how a list of data may be produced through output to XML - such a list may then be used in the browser domain to populate either a drop list or a data grid.

The example uses three Lava system tables (Sys\_Objects, Sys\_Schemas and Sys\_Tables) to produce a list of table attributes for a nominated set of system tables. Several pertinent LavaStream coding techniques are illustrated, including row-level looping through a table, direct relational access to row ID-based relational tables, and the XML-related output techniques.

```
TYPE
  ObjectType          = ROWTYPE
  Object              : STRING[60];
  Schema              : STRING[20];
  Rows                : INTEGER;
  Columns             : INTEGER;
END;

PROCEDURE TableList(pIndex, pCount : INTEGER);
VAR
  SysRow      : LAVA.SYSTEM.Sys_Objects.TYPE;
  SchemaRow   : LAVA.SYSTEM.Sys_Schemas.TYPE;
  TableRow   : LAVA.SYSTEM.Sys_Tables.TYPE;
  GridEntry   : ObjectType;
  Index       : INTEGER;
  rc          : INTEGER;
BEGIN
  rc := pIndex;
  rc := pCount;

  (* Outer tag *)
  OUTPUT(XMLHEADER(TRUE, 'items'));
  OUTPUT(NEWLINE);
  (* Header group *)
  OUTPUT(XMLHEADER(FALSE, 'headers'));
  OUTPUT(NEWLINE);
  (* Grid formatting data *)
  OUTPUT('<element width="50">rowid</element>');
  OUTPUT('<element width="50">#</element>');
  OUTPUT('<element width="200">Table</element>');
  OUTPUT('<element width="150">Schema</element>');
  OUTPUT('<element width="100">Row size</element>');
  OUTPUT('<element width="100">Columns</element>');
  OUTPUT(NEWLINE);
```

## Manual Scope and Target Audience

```
OUTPUT (XMLFOOTER('headers'));
OUTPUT (NEWLINE);

OUTPUT (XMLHEADER (FALSE, 'rowset'));
OUTPUT (NEWLINE);
FOR Index:= pIndex TO pIndex + pCount - 1 DO
  SysRow := LAVA.SYSTEM.Sys_Objects[Index];
  rc := LAVASTATUS();
  CLEAR(GridEntry);
  GridEntry.ROWID := Index;
  IF rc # 0 THEN
    GridEntry.Object := '** no object **';
    GridEntry.Schema := '-';
    OUTPUT (XMLROWDATA('rowdata', GridEntry, TRUE));
    OUTPUT (NEWLINE);
    CYCLE;
  END;
  GridEntry.Object := SysRow.Object_name;
  SchemaRow := LAVA.SYSTEM.Sys_Schemas[SysRow.Schema_id];
  TableRow := LAVA.SYSTEM.Sys_Tables[SysRow.Object_id];
  GridEntry.Schema := SchemaRow.Schema_name;
  GridEntry.Rows := TableRow.RowSize;
  GridEntry.Columns := TableRow.ColumnCount;
  OUTPUT (XMLROWDATA('rowdata', GridEntry, TRUE));
  OUTPUT (NEWLINE);
END;
OUTPUT (XMLFOOTER('rowset'));
OUTPUT (NEWLINE);

OUTPUT (XMLFOOTER('items'));
```

Note that the NEWLINE outputs in the above code are unnecessary, and are merely added to render the output more human-readable.

Given input parameters **1,20** the following output XML is created :

```
<?xml version="1.0" encoding="utf-8" ?> <items>
  <headers>
<element width="50">rowid</element><element width="50">#</element><element
width="200">Table</element><element width="150">Schema</element><element
width="100">Row size</element><element width="100">Columns</element>
</headers>
  <rowset>
    <rowdata ROWID="1">    <Object type="10">Sys_DataFiles</Object>    <Schema
type="10">SYSTEM</Schema>    <Rows type="3">29</Rows>    <Columns
type="3">10</Columns>    </rowdata>
    <rowdata ROWID="2">    <Object type="10">Sys_Tables</Object>    <Schema
type="10">SYSTEM</Schema>    <Rows type="3">70</Rows>    <Columns
type="3">19</Columns>    </rowdata>
    <rowdata ROWID="3">    <Object type="10">Sys_Objects</Object>    <Schema
type="10">SYSTEM</Schema>    <Rows type="3">187</Rows>    <Columns
type="3">27</Columns>    </rowdata>
    <rowdata ROWID="4">    <Object type="10">Sys_Table_Columns</Object>    <Schema
type="10">SYSTEM</Schema>    <Rows type="3">125</Rows>    <Columns
type="3">24</Columns>    </rowdata>
    <rowdata ROWID="5">    <Object type="10">Sys_Schemas</Object>    <Schema
type="10">SYSTEM</Schema>    <Rows type="3">593</Rows>    <Columns
type="3">14</Columns>    </rowdata>
    <rowdata ROWID="6">    <Object type="10">Sys_Parameter</Object>    <Schema
type="10">SYSTEM</Schema>    <Rows type="3">606</Rows>    <Columns
type="3">22</Columns>    </rowdata>
```

## Manual Scope and Target Audience

```
<rowdata ROWID="7">      <Object type="10">Sys_Memory</Object>      <Schema
type="10">SYSTEM</Schema>      <Rows type="3">29</Rows>      <Columns type="3">7</Columns>
</rowdata>
  <rowdata ROWID="8">      <Object type="10">Sys_Map</Object>      <Schema
type="10">SYSTEM</Schema>      <Rows type="3">38</Rows>      <Columns type="3">9</Columns>
</rowdata>
  <rowdata ROWID="9">      <Object type="10">** no object **</Object>      <Schema
type="10">-</Schema>      <Rows type="3">0</Rows>      <Columns type="3">0</Columns>
</rowdata>
  <rowdata ROWID="10">      <Object type="10">Sys_Alias</Object>      <Schema
type="10">SYSTEM</Schema>      <Rows type="3">71</Rows>      <Columns type="3">6</Columns>
</rowdata>
  <rowdata ROWID="11">      <Object type="10">Sys_RefObjectLink</Object>      <Schema
type="10">SYSTEM</Schema>      <Rows type="3">4</Rows>      <Columns type="3">1</Columns>
</rowdata>
  <rowdata ROWID="12">      <Object type="10">Sys_Cache</Object>      <Schema
type="10">SYSTEM</Schema>      <Rows type="3">57</Rows>      <Columns
type="3">13</Columns>      </rowdata>
  <rowdata ROWID="13">      <Object type="10">Sys_ParameterGroup</Object>      <Schema
type="10">SYSTEM</Schema>      <Rows type="3">282</Rows>      <Columns
type="3">8</Columns>      </rowdata>
  <rowdata ROWID="14">      <Object type="10">Sys_ColumnBufferPool</Object>      <Schema
type="10">SYSTEM</Schema>      <Rows type="3">69</Rows>      <Columns
type="3">18</Columns>      </rowdata>
  <rowdata ROWID="15">      <Object type="10">Sys_ParameterType</Object>      <Schema
type="10">SYSTEM</Schema>      <Rows type="3">296</Rows>      <Columns
type="3">10</Columns>      </rowdata>
  <rowdata ROWID="16">      <Object type="10">Sys_Cursors</Object>      <Schema
type="10">SYSTEM</Schema>      <Rows type="3">25</Rows>      <Columns type="3">6</Columns>
</rowdata>
  <rowdata ROWID="17">      <Object type="10">Sys_Event_Module</Object>      <Schema
type="10">Event</Schema>      <Rows type="3">272</Rows>      <Columns type="3">8</Columns>
</rowdata>
  <rowdata ROWID="18">      <Object type="10">Sys_Event_Procedure</Object>      <Schema
type="10">Event</Schema>      <Rows type="3">326</Rows>      <Columns type="3">9</Columns>
</rowdata>
  <rowdata ROWID="19">      <Object type="10">Sys_ColumnValue</Object>      <Schema
type="10">SYSTEM</Schema>      <Rows type="3">94</Rows>      <Columns type="3">9</Columns>
</rowdata>
  <rowdata ROWID="20">      <Object type="10">Sys_Event_Class</Object>      <Schema
type="10">Event</Schema>      <Rows type="3">222</Rows>      <Columns type="3">7</Columns>
</rowdata>
</rowset>
</items>
```