ALPHA SOFTWARE CORPORATION

Xbasic Guide

An introduction to Xbasic

Examples used in this book are fictitious. Names, places, and incidents either are the products of the author's imagination or are used fictitiously. Any resemblance to actual persons, living or dead, events, or locales is entirely coincidental.

First Published: January 13, 2020

Revised Edition: January 27, 2020

Published by Alpha Software Corporation www.alphasoftware.com

Table of Contents

1 Welcome to Xbasic in Alpha Anywhere

Alpha Anywhere is a powerful application development software package. Most of what you need to do can be built in Alpha Anywhere without writing any code. That being said, server-side scripts can extend the core functionality that comes with Alpha Anywhere to incorporate data validation, create workflows, or build complex server-side routines to automate reporting, business processes, and more. In Alpha Anywhere, Xbasic is used to create server-side scripts to perform various tasks.

The Xbasic programming language implements many of the same constructs (including variables, arrays, functions, conditionals, and loops) as other languages, such as C and JavaScript. Xbasic can also interact directly with Node.js modules and C# .NET libraries.

This guide assumes you are already familiar with Alpha Anywhere and understand how to create workspaces and connection strings.

The goal of this guide is to give you a solid grounding in basic programming concepts needed to use Xbasic to customize your Alpha Anywhere applications to your exact business requirements. You can refer to the Alpha Anywhere Documentation to explore everything Xbasic can do. The best way to learn different Xbasic commands and techniques is through experimentation.

1.1 Setting Up Your Workspace

Before you begin, you need to create a new Alpha Anywhere workspace and a connection string named "AADemo-Northwind," which is used later in this guide to interface with the sample Northwind database included in the Alpha Anywhere installation.

1.1.1 Create the Workspace

- Launch Alpha Anywhere
- Create a new Workspace named "XbasicGuide."

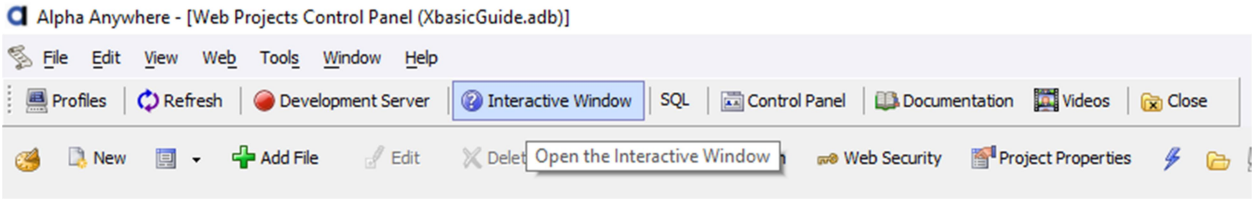
1.1.2 Create the Northwind Demo Connection String

- Open the Web Projects Control Panel
- Select Tools > AlphaDAO Connection Strings
- Click the "Create AADemo-Northwind Connection String" link at the bottom of the window.
- Confirm creating the connection. Then, return to the Web Projects Control Panel.

1.2 The Interactive Window

The Xbasic Interactive Window executes code as you write it, letting you see the results of individual Xbasic commands immediately. The line-by-line interactive nature of the Interactive Window makes it easy to test Xbasic scripts and explore how an Xbasic command works. The Interactive Window is available everywhere you can add Xbasic, usually as a tab in the Xbasic editor.

Let's open the Interactive Window and become familiar with it by executing some simple Xbasic expressions. Click the "Interactive Window" toolbar button on the Web Projects Control Panel to open the Xbasic Interactive Window.



Type the following in the Interactive Window and press enter:

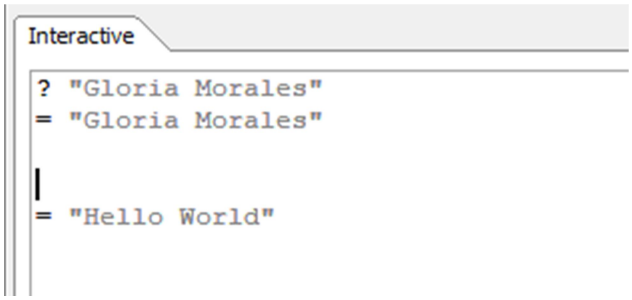
```
? "Hello World"
```

You should see the following output in the Interactive Window:

```
="Hello World"
```

The `?` operator prints the value of an expression in the Interactive Window. The Enter key executes the Xbasic on the line where the text cursor is located.

Try this: change the message from `"Hello World"` to your name. Then, while the mouse cursor is still on the same line as the `?` statement, press enter. This executes the `?` operator again and prints your name immediately after the statement.



Select all of the text in the Interactive Window and delete it. Then, type the following in the Interactive Window and press Enter, replacing `<Your name here>` with your name.

```
name = "<Your name here>"
```

This line creates a character variable called name and assigns it the value of your name. We can verify this by typing the following command to display the value of `name` in the Interactive Window:

```
? name
```

You should see your name output in the Interactive Window.

When you create a variable in the Interactive Window, it is available until you close the Interactive Window or delete the variable using the DELETE statement (we discuss DELETE in a later section.) This means that you can continue to reference a variable even if you delete all of the code in the Interactive Window.

Enter the next line in the Interactive Window and run it:

```
today = now()
```

This line of code does two things. First, the `now()` function is called to get the current date and time. Then, a time variable called `today` is created and assigned the return value of the `now()` function.

You can use the `typeof()` function to determine the data type of a variable:

```
? typeof(today)
```

Executing this statement outputs `T`, which stands for "Time". `T` is the data type of the `today` variable. Other data types include character (`C`), numeric (`N`), and logical (`L`). We discuss data types in depth later in this guide.

Let's continue. Enter the following code on a new line and press enter:

```
dayOfWeek = time("Weekday", today)
```

The statement above creates a variable called `dayOfWeek` and sets the value of the variable to the weekday name. The weekday name is extracted from the `today` variable using the `time()`[1].

The `time()` function converts a time value into a character string. It takes two parameters: a format string and a time value. The format string used here is `Weekday`. `Weekday` returns the full name of the weekday in proper case. There are other formatting options available. For example, run the following statement in the Interactive Window:

```
? time("Month d", today)
```

The statement prints the name of the month followed by the day of the month. Many formatting options exist for formatting date and time values. You can learn more about what format options are

[1] https://documentation.alphasoftware.com/index?search=api%20time%20function

available in the online Alpha Anywhere documentation. See Date and Time Format Elements[2] to learn more.

Enter the following in the Interactive Window, executing each line as you write it:

```
message = "Hello " + name + "." + crlf()
message = message + "Today is " + dayOfWeek + "."
? message
```

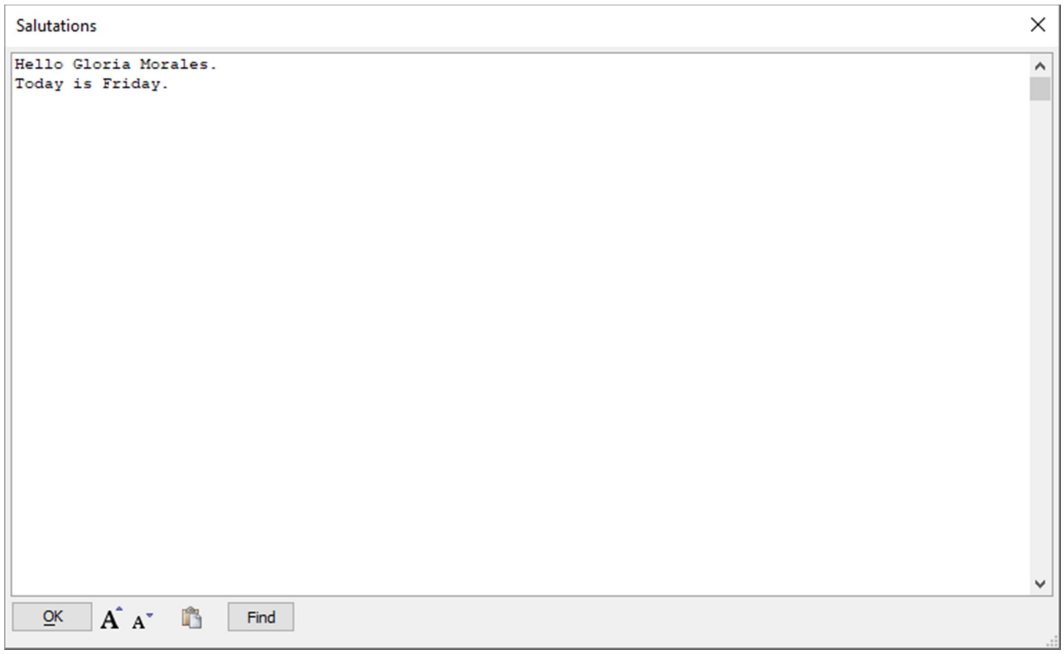
The `+` operator is used to concatenate multiple strings together into a single character string. The result is stored in the `message` variable. The second line adds the day of the week to the message. The `crlf()` function adds a newline to the message, breaking the text into two lines.

```
message = "Hello " + name + "." + crlf()
message = message + "Today is " + dayOfWeek + "."
? message
= Hello Gloria Morales.
Today is Friday.
```

There are other ways to inspect the value of a variable in the Interactive Window. One of them is using the `showvar()` function. `showvar()` displays a variable in a popup window. Let's display the message variable in a popup window. Type the code below into the Interactive Window and press Enter:

```
showvar(message,"Salutations")
```

This statement displays the `message` variable in a window with the title `"Salutations"`:



[2] https://documentation.alphasoftware.com/index?search=api%20date%20and%20time%20format%20elements

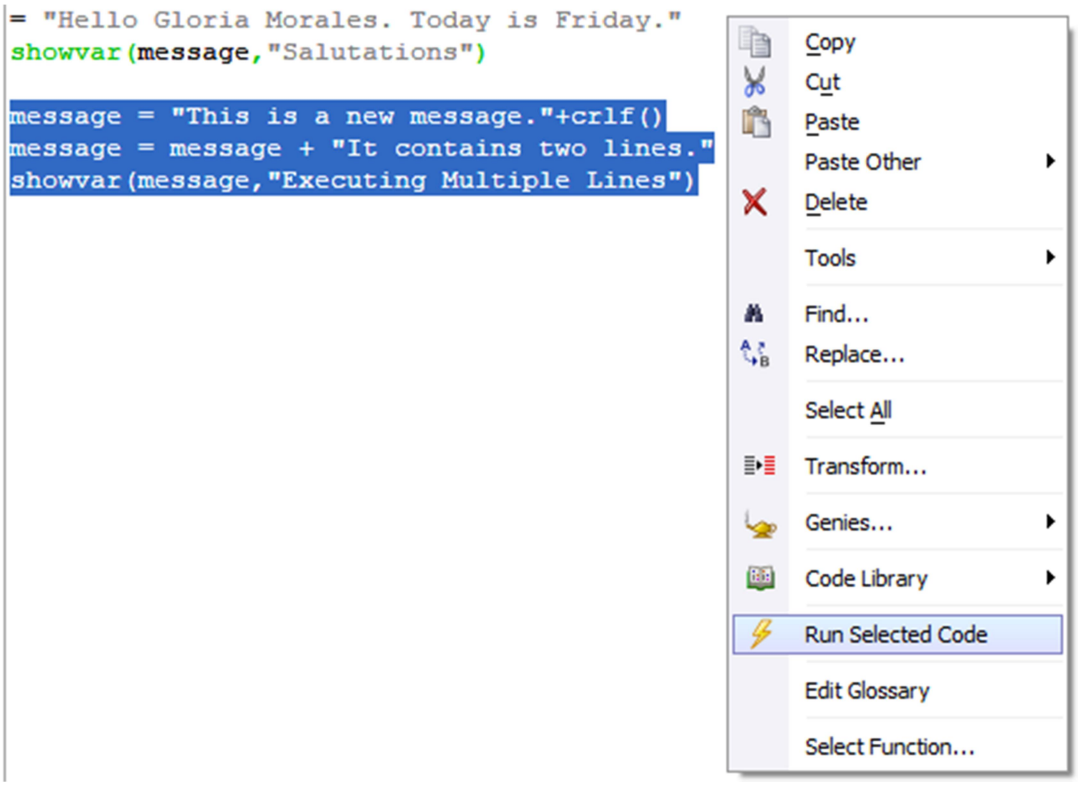
1.2.1 Interactive Window Commands

The Interactive Window has several commands for running and modifying code in the window. You are already familiar with two: executing the current line using the Enter key and printing the value of an expression using the `?` operator.

1.2.1.1 Run Selected

Multiple lines of code can be executed by highlighting multiple lines of code then right-clicking to open the context menu and selecting "Run Selected Code". Try it out: copy and paste the following code into the Interactive Window. Then select it, right-click anywhere in the whitespace of the Interactive Window, and select Run Selected Code.

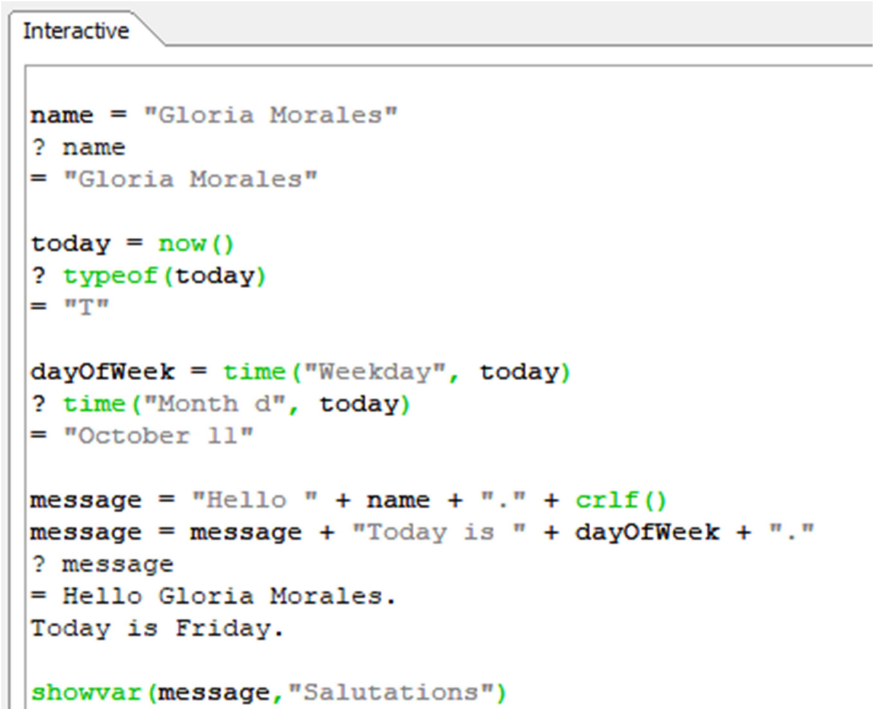
```
message = "This is a new message."+crlf()
message = message + "It contains two lines."
showvar(message,"Executing Multiple Lines")
```



1.2.1.2 Insert Newline

Holding the Ctrl key while pressing Enter adds a new line to the Interactive Window without executing any statements. Inserting newlines without executing code is useful when writing multi-line statements, such as an `IF` statement, or in cases where you would like to insert a newline within existing statements.

Place your mouse cursor at the beginning of the first line of the Interactive Window. Then press enter while holding down the Ctrl key.



1.2.1.3 Clear Screen

Typing `CLE` and then pressing enter on a blank line deletes all text in the Interactive Window at and below the line containing the `CLE` command.

Place the cursor on the first line of the Interactive Window. Then, type `CLE` and press Enter.

Deleting the text in the Interactive Window using the `CLE` command does not delete variables created during the Interactive Window session.

2 The Xbasic Programming Language

2.1 Variables & Data Types

Variables store values for later use. Before a variable is used either by a script or in an expression, it must be declared. A variable is declared either explicitly as a formal declaration in a script, a table, a set, a form, or an application; or implicitly by assigning it a value for the first time in a script. In general, a variable declaration must specify two things: the name of the variable, and the type of data the variable can contain.

A variable name must start with a letter (`A` to `Z`, `a` to `z`). Subsequent characters can be alphanumeric or an underscore (`A` to `Z`, `a` to `z`, `0` to `9`, `_`). Variable names are not case-sensitive, so the names `GONZO` and `Gonzo` refer to the same thing. Variable names cannot contain spaces. Xbasic does not prevent you from using a keyword as a variable name; however, you should prefer to avoid using keywords as variable names in your scripts. A list of keywords can be found in "Xbasic Keywords" in the Appendix.

A variable is implicitly defined by assigning it a value with the assignment operator (`=`), using it in a `FOR` loop statement, or declaring it as a parameter of a function.

```
VariableName = Expression
VariableName[Subscript] = Expression
FOR VariableName = StartValue TO EndValue
FUNCTION FunctionName AS DataType (Parameter AS DataType, Parameter AS
DataType,...)
```

For example:

```
x = 3
arr[2] = "Orange"
FOR index = 1 TO 10
```

It is often important to explicitly define a variable before using it in a script. The `DIM` statement explicitly defines variables:

```
DIM variable_name AS data_type
```

For example:

```
DIM x AS N 'a numeric variable
DIM arr[0] AS C 'a character array
DIM args AS SQL::Arguments 'a SQL Arguments object
```

An explicit declaration is necessary if it is ambiguous where the variable is used or if Xbasic Strict Mode is enabled. When strict mode is enabled, all variables must be declared before they can be used.

The `DIM` statement is also necessary if you wish to create a complex variable, such as an array or object pointer variable.

2.1.1 Data Types

In the statement `DIM variable_name AS data_type`, the *`data_type`* declares the type of data stored in the variable. Xbasic data types can be simple types, such as character, logical, or numeric, or more complex types such as arrays, collections, objects, or function pointers. You can also declare that a variable can contain any value.

Symbol	Name	Description
A	Any	The variable can contain any data type
B	Blob	Binary data
C	Character	Alpha-numeric characters
D	Date	A date between 00/00/0000 and 12/31/9999
F	Function Pointer	Contains a pointer to a function name
L	Logical	True (.T.) or false (.F.)
N	Numeric	A number with a length up to 19 digits
P	Object Pointer Variable or "Dot Variable"	A reference (pointer) to an object, or a pointer to a "dot" variable
T	Time. A date/time value.	A time value that stores the date, hour, minute, seconds, and hundredths of a second
U	Collection	Any data type, depending on what the collection contains

In addition to the basic types listed above, you can also declare class variables. Class variables have methods and types. The Alpha Anywhere Xbasic library includes a wide variety of classes for performing tasks, such as manipulating JSON data, calling REST services, performing SQL queries, and more. A variable declared as a class type uses the `DIM` statement. For example:

```
DIM conn AS SQL::Connection
DIM args AS SQL::Arguments
```

2.1.2 Redeclaring Variables

Once a variable's type is declared, you cannot change it. Attempting to assign a value to a variable that is not the same type as the variable causes an Xbasic error. For example, run the following Xbasic in the Interactive Window:

```
DIM myVar AS N
myVar = 67

myVar = now()
```

Note that when you execute the `myVar = now()` statement, an error appears in the Interactive Window:

```
ERROR: Variable type mismatch: Cannot assign data of type 'T' to variable of type
'N'.
```

Variables cannot be assigned values of another type. If you want to change the data type stored by a variable, the variable must first be deleted using the `DELETE` statement. After deleting the variable, you can create it again with a different data type. EG:

```
DELETE myVar
DIM myVar AS T
myVar = now()
```

When working with large variables, it can be beneficial to `DELETE` them when they are no longer needed. Loading files from disk, SQL query results, and data returned from web services can consume large amounts of memory.

Deleting variables is also useful when working in the Interactive Window to "reset" variables to an undefined state, ensuring that the variables contain the data you expect. For example, enter the following statements in the Interactive Window:

```
DIM args AS SQL::Arguments
args.set("City","Madrid")
args.set("Country","Spain")
? args.find("City")

DELETE args

DIM args AS SQL::Arguments
args.set("Country","Spain")
? args.find("City")
```

The first time you execute `? args.find("City")`, the following object is output in the Interactive Window:

```
= Data = "Madrid"
IsNull = .F.
Name = "city"
Usage = 0
XML = <SQLArgument>
    <Name>city</Name>
    <Data Type="C">Madrid</Data>
    <IsNull Type="L">0</IsNull>
    <Usage>Input</Usage>
</SQLArgument>
```

The second time you execute `? args.find("City")`, you see the following statement:

```
= <No data returned>
```

`DELETE args` deleted the `args` variable. When the `args` variable was then recreated using the `DIM` statement, the "City" argument was never created using the `set()` method of the `args` variable.

2.1.3 Converting Data Types

A value can be converted to other types using the `convert_type()` function.

```
DIM quantity AS N
quantity = convert_type("123","N")
```

`convert_type()` converts the value of a variable into the requested type. If data value cannot be converted to the requested type, the function will return a character value.

Values can also be implicitly converted to character strings using the concatenation operator (`+`). For example, run the code below in the Interactive Window:

```
DIM price AS N = 1.5
DIM quantity AS N = 10
DIM total AS N
total = price * quantity
? typeof(total)
totalStr = "" + total
? typeof(totalStr)
```

Specific functions exist for converting data to formatted character strings and converting character strings to other data types, such as date and time values. For example, the `time()` function converts a date or time value to a formatted character string.

For more information about converting character variables, see Character Conversion Functions[3] in the Alpha Anywhere documentation.

2.1.4 Default Values

Variables can be assigned a default value when they are declared. Default values are used in cases where a variable may or may not exist but is required later in the script. Defining default values is also useful when working with session variables in web applications (we discuss session variables later in this guide.)

To specify a default value for a variable, use the `DEFAULT` keyword when the variable is declared:

```
DIM name AS C = DEFAULT "Steve"
```

Using the `DEFAULT` keyword to assign the initial value to a variable is similar to using the assignment operator (e.g., `DIM name AS C = "Steve"`) to set the variable's value with one major difference: if the variable already exists, the `DIM AS DEFAULT` statement is ignored.

When declaring a variable with the `DEFAULT` keyword, Alpha Anywhere first checks to see if the variable exists. If the variable does not exist, the variable is created and assigned the specified `DEFAULT` value. If the variable already exists, however, the variable the existing variable is not modified.

The Interactive Window is a useful tool to help understand how the `DEFAULT` keyword works. Copy the following code into the Interactive Window, replacing `<Your name here>` with your name:

```
name = "<Your name here>"
today = date()
dayOfWeek = time("Weekday", today)

DIM name AS C = DEFAULT "Susan"

message = "Hello " + name + ". "
message = message + "Today is " + dayOfWeek + "."

showvar(message,"Salutations")
```

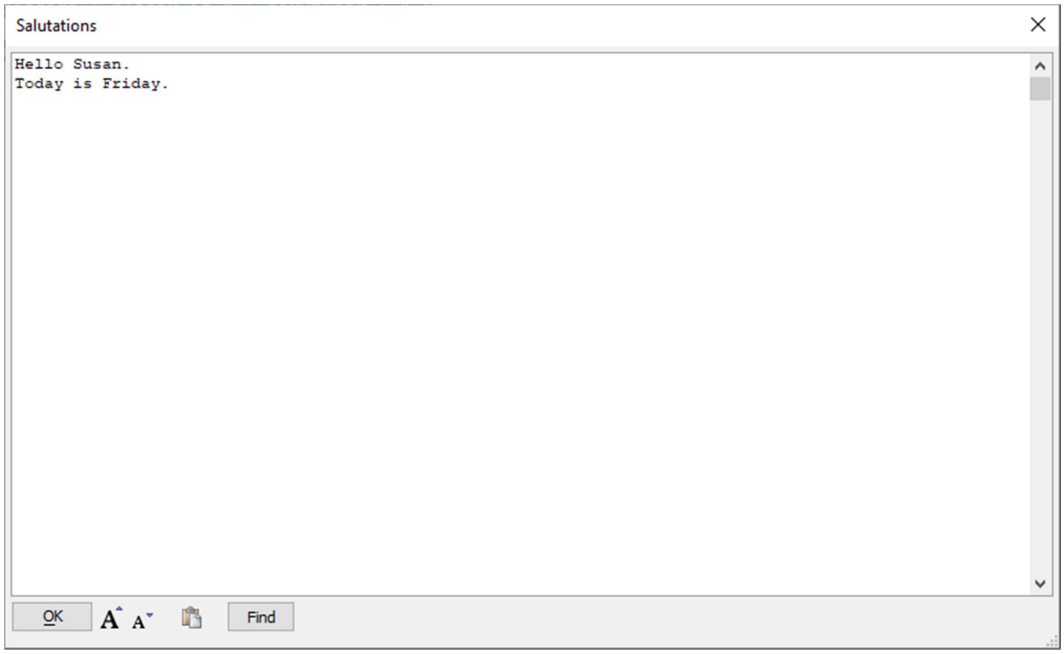
Run the script by selecting all of the code and using the "Run Selected Script" tool from the right-click menu. When the script executes, it creates a variable called `name` and sets the value of the variable to your name. When the `DIM name AS C = DEFAULT "Susan"` statement is processed, Alpha Anywhere ignores the statement because the variable `name` already exists and has a value.

Now, change the first line to the following:

[3] https://documentation.alphasoftware.com/index?search=api%20character%20conversion%20function

```
DELETE name
```

Rerun the script. This time, when the script executes, it deletes the `name` variable. When the `DIM name AS C = DEFAULT "Susan"` statement executes, the `name` variable doesn't exist, so it is created and assigned the value "Susan".



2.2 Expressions

An expression is a combination of variables, constants, operators, and functions that evaluate to another value. For example, in the code below, the text to the right of the equal (=) sign is an expression:

```
Name = Firstname + " " + Lastname
```

2.2.1 Operators

An operator is a symbol used to represent a mathematical, relational, logical, or string operation. For example, the asterisk (*) character represents multiplication. Functions, field names, and constants can be combined with operators to form complex expressions. Most operators use the following syntax:

```
expression operator expression
```

We have used several operators and functions in this guide already, including the assignment (=) operator to assign values to variables, the concatenation (+) operator to create character strings by combining variables (`name`, `dayOfWeek`) and string constants (`"Hello "`, `"Today is "`), and the `?` output operator to display the values of variables and functions in the Interactive Window.

```
result = 12 * 7
? result
= 84
```

Expressions execute in the order of Precedence. Operator Precedence defines the order in which operators in an expression execute. In general, operators are evaluated in the following order:

- Parentheses and Function calls
- Negation (`.NOT.`)
- Exponentiation (`^`, `**`)
- Multiplication, Division, and Substring Inclusion (`*`, `/`, `$`, `!$`)
- Addition and subtraction (`+`, `-`)
- Relative and relative or equal (`<=`, `>=`, `<`, `>`)
- Equal, Exactly Equal, Not Equal, and Not Exactly Equal (`=`, `==`, `<>`, `!=`)
- And and Exclusive Or (`.AND.`, `.XOR.`)
- OR (`.OR.`)

Including operators in order according to their precedence allows implied parentheses when combining operations. For example:

```
A >= 5 .AND. B = 10 .OR. A >= 50
(A >= 5 .AND. B = 10) .OR. A >= 50
```

Because the `.AND.` operator has precedence over the `.OR.` operator, both statements are equivalent. When multiple operations have the same precedence, they evaluate from left to right. For example:

```
? 0.6 < 10 .AND. "Apple"="Apple" .AND. "Alpha" < "Beta"
= .T
```

2.2.1.1 Mathematical Operators

Mathematical operators are used in numeric, date, or character expressions to yield results of the same type.

Operator	Description	Example
()	Parentheses. Used to group operations	? (3 + 2) * 5 = 25
^	Exponentiation.	? 10^2 = 100
******	Exponentiation.	? 7**2 = 49
*****	Multiplication.	? 8 * 8 = 64
/	Division.	? 144 / 12 = 12
+	Addition. Adds two numbers, dates, or time values together. When used with characters, the two variables are concatenated.	? 1+3 = 4 ? {1/23/2001}+30 = {02/22/2001} ? {08/19/2019 03:14:58 77 pm} + 32000 = 08/20/2019 12:08:18 77 am ? "Welcome" + " " + "Home" = "Welcome Home"
-	Subtraction. Subtracts a number from another number, days from a date, or seconds from a time or date-time value. When used with character strings, concatenates two character values together, removing trailing whitespace from the first character string.	? 12 - 7 = 5 ? {3/15/2001} - 90 = {12/15/2000} ? {03:14:58 77 pm} - 3600 = 02:14:58 77 pm ? "race " - "car" = "racecar"

2.2.1.2 Comparison Operators

Comparison operators compare two expressions which must be of the same type (either character, numeric, or date) and returns a result of true (`.T.`) or false (`.F.`). Any expression involving a comparison operator is called a logical expression. Comparison operators are typically used in creating search criteria, filters, or are used with the `IF` and `CASE` functions. All comparison operators, except for substring inclusion, can evaluate numeric, date, or character values. For date values, earlier dates have lower values. For character values, the character's corresponding ASCII value is used in the comparison.

Operator	Description	Example

=	Equals. When used in a logical expression, compares two values for equality. If used with character strings, removes all trailing whitespace and performs a case-insensitive comparison.	`? 7 = "7"`
		`= .F.`
		`? 7 = 4+3`
		`= .T.`
		`? "hello  " = "HELLO"`
		`= .T.`

==	Exactly Matching. When used in a logical expression, compares two values for equality. When used with character strings, performs a case-sensitive comparison. Does not remove whitespace.	`? 7 == "7"`
		`= .F.`
		`? 7 == 4+3`
		`= .T.`
		`? "hello  " == "HELLO"`
		`= .F.`

>	Greater Than.	`? 7 > 7`
		`= .F.`
		`? {01/01/2020} > now()`
		`= .T.`

>=	Greater Than or Equal To.	`? 7 >= 7`
		`= .T.`
		`? {01/01/2020} >= now()`
		`= .F.`

<	Less Than.	`? {01/01/2020} < now()`
		`= .F.`
		`? 10 < 12`
		`= .T.`

<=	Less Than or Equal To.	`? 7 <= 6`
		`= .F.`
		`? {06/12/2019} <= date()`
		`= .T.`

<>	Not Equal. When used with character strings, the <> operator removes trailing whitespace from both strings and performs a case-insensitive comparison.	`? 2 <> 3`
		`= .T.`
		`? "Foley " <> "foley    "`
		`= .F.`
		`? "Foley" <> "folley"`
		`= .T.`

Operator	Description	Example

!=	Not Exactly Matching. When used with character strings, performs a case-sensitive comparison. Does not remove whitespace.	`? 2 != 3`
		`= .T.`
		`? "Foley " != "foley "`
		`= .T.`
		`? "Foley" != "folley"`
		`= .T.`

$	Contains Substring. Performs a case-insensitive test to determine if the first character string is found in the second character string.	`? "CARS" $ "racecars"`
		`= .T.`
		`? "cars" $ "racecars"`
		`= .T.`

$$	Contains Substring. Performs a case-sensitive test to determine if the first character string is found in the second character string.	`? "CARS" $$ "racecars"`
		`= .F.`
		`? "cars" $$ "racecars"`
		`= .T.`

!$	Does Not Contain Substring. Performs a case-insensitive test to determine if the first character string does not exist in the second character string.	`? "cats" !$ "racecars"` `= .T.`

2.2.1.3 Logical Operations

Logical operators are used between logical expressions (two expressions that return a `.T.` or `.F.` value) to yield logical results.

Operator	Description	Example

=	Equals.	`? .T. = .T.`
		`= .T.`
		`? .F. = .F.`
		`= .T.`
		`? .F. = .T.`
		`= .F.`

.AND.	Logical AND.	`? .T. .AND. .F.` `= .F.`
.OR.	Logical OR.	`? .T. .OR. .F.` `= .T.`

.XOR.	Logical XOR.	`? .T. .XOR. .F.`
		`= .T.`
		`? .T. .XOR. .T.`
		`= .F.`
		`? .F. .XOR. .F.`
		`= .F.`

.NOT.	Negation.	`? .NOT. (4 == 5)` `= .T.`

2.2.2 Delimiters

Constant values can be assigned to character strings using one of two methods: double-quotes (") and delimiters. When declaring multi-line character strings with double-quotes, you must concatenate additional lines using a combination of the `crlf()` function to insert a new line and the concatenation (+) operator. For example:

```
DIM str AS C
str = "This is the first line." + crlf()
str = str + "This is the second line." + crlf()
str = str + "This is the last line."
```

This example can be simplified by using delimiters. Delimiters are used to define multi-line character strings. A character string declared using delimiters begins with the `<<%delimiter%` statement and ends with the `%delimiter%` statement. The text, `delimiter`, can be any value you desire as long as it is identical in the beginning and ending delimiter operator. For example, we can recreate the previous example using delimiters as follows:

```
DIM str AS C
str = <<%myStr%
This is the first line.
This is the second line.
This is the last line.
%myStr%
```

IMPORTANT: `<<%DELIMITER%` must be terminated with a newline. Any spaces or other characters after the closing `%` will result in an "Extra characters at end of expression" error.

The Xbasic auto help system provides suggestions for delimiters as you write your scripts. There are several special case delimiters, such as `%code%` and `%html%`, which add unique behaviors such as syntax highlighting and code validation. The delimiters offered through auto help include:

Delimiter	Description
<<%code% %code%	Xbasic code string. Xbasic written inside code blocks will have full access to auto help and includes syntax highlighting.
<<%css% %css%	CSS code string. CSS written inside a CSS block will include syntax highlighting and auto help.
<<%html% %html%	HTML code string. HTML written inside html blocks will have syntax highlighting and auto help.
<<%js% %js%	JavaScript code string.
<<%json% %json%	JSON string.
<<%xml% %xml%	XML string. XML code written inside xml blocks will have syntax highlighting and auto help.

Delimiter	Description
<<%str% %str% <<%txt% %txt%	Text string. No special formatting is applied.

2.2.3 Comments

You can add comments to Xbasic by starting a line with an apostrophe (`'`). For example:

```
'This is a comment
```

Comments are useful for describing the purpose or behavior of an Xbasic script or function. All text in a comment is ignored when an Xbasic script executes. Comments are used in examples throughout this guide to explain what a script does.

```
'Square the value
square = value * value

'Create a list of countries
DIM europe AS C = "Denmark,Norway,Sweden"
```

Comments can also be used to disable code that you don't want to execute but would like to keep in your script:

```
'DIM field_value AS D
'DELETE field_value
```

2.3 Conditional Statements

Conditional statements execute code blocks when a specific condition is met.

2.3.1 IF statements

The most common and useful conditional statement in the Xbasic language uses the `IF ... THEN ... ELSE ... END IF` syntax. The statement begins by testing whether an expression is true. If the expression is true (`.T.`), the statement after the `THEN` clause executes. If the expression is false (`.F.`), the statement after the `ELSE` clause is executed.

For example, copy the code below into the Interactive Window and run it:

```
today = now()
weekday = dow(today)
IF (weekday = 1 .OR. weekday = 7) THEN
    msg = "Business is closed."
ELSE
    msg = "Business is open."
END IF
showvar(msg,"Open or Closed?")
```

In the example above, the `now()` function is used to get the current date. It then uses the `dow()` function to get the weekday. The `weekday` is then tested to determine what message to display. If it is the weekend (Sunday (1) or Saturday (7)), then the business is closed. Otherwise, the business is open.

`ELSE IF` can be used to create multiple test cases in an `IF` statement. If the `IF` statement contains multiple tests, the code following the first expression that evaluates to `.T.` executes. If no `IF...ELSE IF` tests evaluate to `.T.`, the `ELSE` condition is executed. Run the code below in the Interactive Window:

```
IF (weekday = 1 .OR. weekday = 7) THEN
    msg = "No work today! It is the weekend."
ELSE IF (weekday = 2) THEN
    msg = "It is the first day of the work week."
ELSE IF (weekday = 6) THEN
    msg = "It is the last day of the work week."
ELSE
    msg = "It is the middle of the work week."
END IF
showvar(msg,"Work week")
```

If `weekday` is 1 (Sunday) or 7 (Saturday), the script displays the message that it is the weekend. Otherwise, if `weekday` is 2 (Monday), then a message is shown stating that it is the first day of the work week. Otherwise, if `weekday` is 6 (Friday), then the message states it is the last day of the work week. If `weekday` is any other value, then the `ELSE` condition is executed, and we see a message that it is the middle of the work week.

2.3.2 SELECT Statements

`IF...THEN...ELSE` statements become tedious when you have more than two alternatives. The `SELECT...CASE` language element provides a much easier way to test for multiple conditions. The `SELECT...CASE` statement allows you to test any number of expressions. The statements immediately following the first expression that evaluates to `.T.` are executed up to, but not including, the next `CASE` statement. Run the following code in the Interactive Window:

```
today = now()
weekday = dow(today)
msg = ""
SELECT
    CASE weekday = 1
        msg = "Today is Sunday"
    CASE weekday = 2
        msg = "Today is Monday"
    CASE weekday = 3
        msg = "Today is Tuesday"
    CASE weekday = 4
        msg = "Today is Wednesday"
    CASE weekday = 5
        msg = "Today is Thursday"
    CASE weekday = 6
        msg = "Today is Friday"
    CASE weekday = 7
        msg = "Today is Saturday"
END SELECT
showvar(msg,"Day of Week")
```

2.4 Loop Statements

A statement that executes the same block of code multiple times is called a loop statement. Loops are used to perform some action for multiple entries in an array or while waiting for a response to an asynchronous action, such as Queues or callbacks.

Loops are created in Xbasic using `FOR` and `WHILE` statements.

2.4.1 FOR Loops

An Xbasic `FOR` loops is used to repeat a block of code a specified number of times. The `FOR` loop statement includes a numeric counter, a start value, and an end value. Each time the `FOR` loop executes, the counter is increased by 1.

```
DIM countingMsg AS C = ""
FOR counter = 1 TO 10
    'Count to 10 in the trace log
    countingMsg = countingMsg + counter + crlf()
NEXT
showvar(countingMsg,"Counting from 1 to 10")
```

You can optionally specify a `STEP` value if you wish to increment the counter by a value that is not 1.

```
FOR counter = start TO end STEP increment
    'Xbasic code to execute
NEXT
```

The `STEP` value can be any numeric number. For example, you can count backward from 10 to 1 by specifying a `STEP` of -1:

```
countingMsg = ""
FOR counter = 10 TO 1 STEP -1
    'Count from 10 to 1
    countingMsg = countingMsg + counter + crlf()
NEXT
showvar(countingMsg,"Counting Backwards")

countingMsg = ""
FOR counter = 0 TO 10 STEP 2
    'Count from 0 to 10 by 2
    countingMsg = countingMsg + counter + crlf()
NEXT
showvar(countingMsg,"Counting by 2s")
```

You can optionally include the variable name for the counter in the `NEXT` statement for the `FOR` loop.

```
countingMsg = ""
DIM counter AS N
FOR x = 1 TO 10
    FOR y = 1 TO 10
        'Count to 100
        counter = 10 * (x - 1) + y
        countingMsg = countingMsg + counter + crlf()
    NEXT y
NEXT x
showvar(countingMsg,"Counting from 1 to 100")
```

It is not required to include the counter variable in the `NEXT` statement. However, for long scripts or scripts with multiple `FOR` loops, it is useful to include the counter to document which `FOR` loop matches the `NEXT` statement.

2.4.2 FOR EACH: Looping Over Lists, Collections, or Arrays

The `FOR EACH` loop is a type of `FOR` loop for iterating over a data set stored in a character list, collection, or array. Instead of using a counter to track progress through the loop, the `FOR EACH` loop iterates over entries in an object.

```
FOR EACH element IN object
    ' Xbasic code to execute
NEXT
```

The value of the current entry in the object can be accessed using the `value` property of the element.

```
valueOfElement = element.value
```

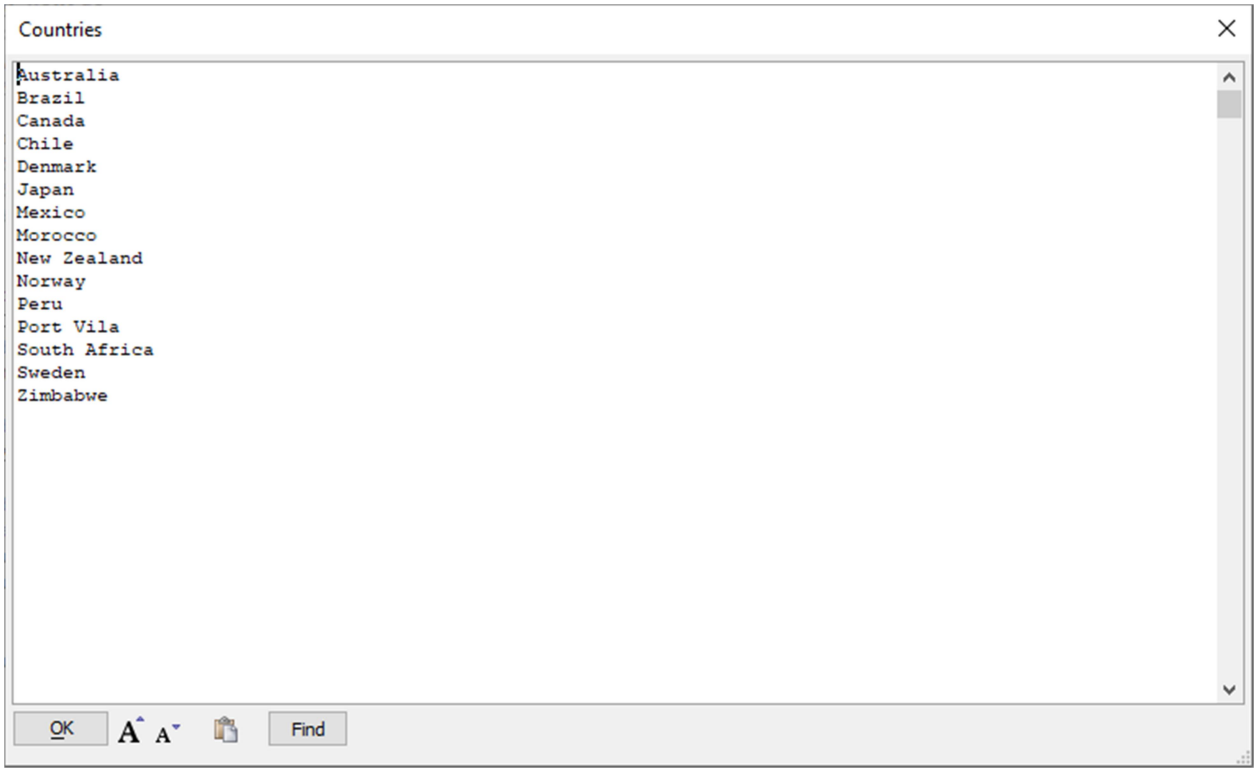
For example, consider the following list of cities and counties:

```
DIM places AS C = <<%txt%
Canberra,Australia
Brasilia,Brazil
Ottawa,Canada
Santiago,Chile
Copenhagen,Denmark
Tokyo,Japan
Mexico City,Mexico
Rabat,Morocco
Wellington,New Zealand
Oslo,Norway
Lima,Peru
Vanuatu,Port Vila
Cape Town,South Africa
Stockholm,Sweden
Harare,Zimbabwe
%txt%
```

The following `FOR EACH` loop builds a list of `countries` from the `places` list:

```
DIM countries AS C
FOR EACH place IN places
    'Get the value of the current element
    currentPlace = place.value

    'Get the country name
    country = word(currentPlace,2,",")

    'Add the country name to the countries list
    'if the country is not in the list yet
    IF (country !$ countries) THEN
        countries = countries + country + crlf()
    END IF
NEXT
showvar(countries,"Countries")
```

When we run the script, the country list is displayed:



2.4.3 Skipping FOR Loop Iterations

It may not be necessary to execute a `FOR` loop until the loop's terminating condition evaluates to true (`.T.`). In some cases, you may only need to execute the code in the loop for some of the elements in the object. Iterations in a loop can be skipped using the `CONTINUE` keyword.

For example, using the `places` list created earlier, we could write the following script to create a list of countries that are not in North America:

```
'Create a list of places that are not in North America:
DIM nonNorthAmericaPlaces as C
DIM northAmerica as C = "USA,Canada,Mexico"
FOR EACH place IN places
    country = word(place.value,2,",")
    'Check to see if country is in Europe
    IF (country $ northAmerica)
        CONTINUE 'place is in North America, skip rest FOR loop
    END IF
    'Add the place to the nonNorthAmericaPlaces list
    nonNorthAmericaPlaces = nonNorthAmericaPlaces + place.value + crlf()
NEXT
showvar(nonNorthAmericaPlaces,"Places not in North America")
```

If the `country` name is in the list of North American countries (`northAmerica`), the `CONTINUE` statement is executed, skipping the remainder of the current iteration of the `FOR EACH` loop and moving to the next entry in the `places` list.

2.4.4 Exiting FOR and FOR EACH Loops

You can exit a `FOR` loop or `FOR EACH` loop at any time using the `EXIT FOR` statement. `EXIT FOR` terminates the `FOR` loop and proceeds to the next line of code after the `NEXT` statement. For example, the script below searches the list of places for the city, Copenhagen. Once the city is found, there is no need to continue searching the rest of the list of places, so the `FOR` loop terminates with the `EXIT FOR` statement:

```
'Find the Country for "Copenhagen"
DIM country AS C
FOR EACH place IN places
    currentPlace = place.value
    city = word(currentPlace,1,",")
    IF (city = "Copenhagen")
        'Country found
        country = word(place.value,2,",")
        EXIT FOR 'Exit the FOR loop because we found the country
    END IF
NEXT
showvar(country,"Copenhagen is located in...")
```

2.4.5 WHILE Loops

`WHILE ... END WHILE` is a control structure that repeats the statements it contains while the Logical Expression evaluates to true (`.T.`). Execution resumes at the line following the `END WHILE` statement when the Logical Expression is false (`.F.`) or when the `EXIT WHILE` statement executes.

```
countingMsg = ""
counter = 1
WHILE counter <= 100
    countingMsg = countingMsg + counter + crlf()
    'Increment counter by 1
    counter = counter + 1
END WHILE
showvar(countingMsg,"Counting from 1 to 100")
```

2.4.6 Exiting WHILE Loops

Similar to `FOR` and `FOR EACH` loops, you can terminate `WHILE` loops early using the `EXIT WHILE` statement. For example, the code below computes the date of the next Tuesday for the current month. If no Tuesday is found before the end of the month (the value of the current month does not match the month of the `nextDay` variable), the `WHILE` loop terminates, and a message is shown stating that there are no Tuesdays left in the month.

```
noMoreTuesdays = .F.
today = now()
nextDay = today + 1
currentMonth = month(today)

WHILE 3 != dow(nextDay) ' 3 = Tuesday
    IF (currentMonth <> month(nextDay)) THEN
       ' The month changed; there are no more Tuesdays
       ' Exit the WHILE loop
        noMoreTuesdays = .T.
        EXIT WHILE
    END IF
    nextDay = nextDay + 1
END WHILE

IF (noMoreTuesdays) THEN
    showvar("There are no more Tuesdays this month.")
ELSE
    showvar("The next Tuesday is on " + nextDay)
END IF
```

2.5 Functions

Functions are named reusable Xbasic code blocks. Functions can be called in expressions or by themselves to perform a task. They can optionally take one or more input arguments (also called parameters) and return a value.

Functions are declared as follows:

```
FUNCTION function_name AS return_type ()
     ' Xbasic code to execute here
END FUNCTION
```

The *return_type* declares the data type of the return value from the function. If a function returns no value, use the void (V) return type. For example:

```
FUNCTION myFunction AS V ()
     ' Xbasic code to execute here
     showvar("Function 'myFunction' has been called","myFunction")
END FUNCTION
```

To call the function, type the function name followed by opening and closing parentheses:

```
myFunction() 'Call myFunction
```

2.5.1 Passing Data to Functions

Xbasic functions can take one or more arguments. Arguments are defined in the parentheses of the FUNCTION declaration:

```
FUNCTION myFunction AS V (name AS C, address AS C, age AS N)
     ' Xbasic code to execute here
     DIM msg AS C = name + ", age " + age + ", lives at " + address + "."
     showvar(msg,"myFunction")
END FUNCTION
```

When the function is called, the arguments are passed as a comma-delimited list inside the parentheses. For example:

```
myFunction("Susan","123 Baker St, Boston, MA", 37)
```

Arguments can be constant values, as shown above, or variables:

```
name = "Susan"
address = "123 Baker St, Boston, MA"
age = 37

myFunction(name,address,age)
```

2.5.2 Declaring Optional Arguments

Arguments can be assigned default values, making them optional. For example:

```
FUNCTION myFunction as V (name as C, address as C, age as N, phoneNumber = "")
     ' Xbasic code to execute here
     DIM msg AS C = name + ", age " + age + ", lives at " + address + "."
     if (len(alltrim(phoneNumber)) > 0) then
          msg = msg + " Phone: " + phoneNumber
     end if
     showvar(msg,"myFunction")
END FUNCTION
```

The assignment operator (`=`) is used instead of the `AS` keyword when declaring arguments with default values. The default value's type determines the argument's type. For example, `phoneNumber` is a character type because it is assigned an empty string.

Optional arguments must be declared after all required arguments.

When calling a function, you can omit optional arguments. For example, in the code below the first line omits the phone number argument when calling `myFunction` while the second line includes phone number:

```
myFunction("Susan","123 Baker St, Boston, MA", 37)
myFunction("Steve","1314 W Elm, Springfield, IL", 44, "123-445-6778")
```

2.5.3 Returning Values

Data is returned from an Xbasic function by assigning an expression to the function name. For example:

```
FUNCTION square AS N (value as N)
    square = value * value
END FUNCTION

square(12)
```

Multiple values can be returned from a function by declaring the function's type as `P`. For example:

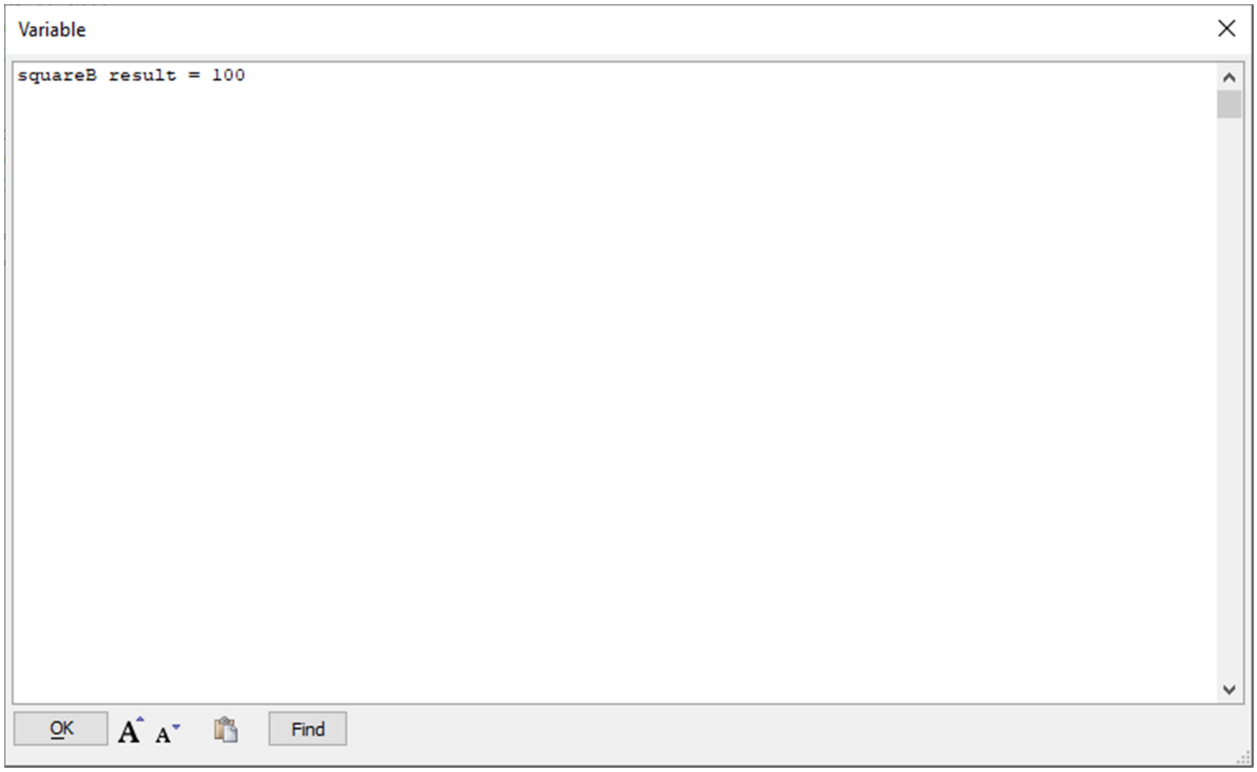
```
FUNCTION getCat AS P ()
    DIM cat AS P
    cat.pet_name = "Savvy"
    cat.breed = "Tuxedo"

    getCat = cat
END FUNCTION

DIM cat AS P
cat = getCat()
showvar(cat)
```

2.5.3.1 Using the RETURN Keyword

Data can alternatively be returned from a function using the `RETURN` keyword. The difference between assigning a value to the function name and `RETURN` is that `RETURN` immediately exits the function. Assigning an expression to the function name does not terminate the function. For example, copy the code below into the Interactive Window and run it.

```
FUNCTION squareA AS N (value AS N)
    DIM result AS N = value * value
    RETURN result
    showvar("squareA result = " + result) ' this code is never executed
END FUNCTION

FUNCTION squareB AS N (value AS N)
    squareB = value * value
    showvar("squareB result = " + squareB) ' this code is always executed
END FUNCTION

aSquare = squareA(10)
bSquare = squareB(10)
```

Note that the `showvar()` statement in `squareA()` never executes, but the `showvar()` statement in `squareB()` runs, displaying the message shown below.



2.5.4 Returning Data Using Arguments

Data can also be returned using function arguments. If an argument is declared as `BYREF`, any modifications to the argument by the function changes the variable passed into the function from in the calling script.

```
FUNCTION squareC AS V (value AS N, BYREF result AS N)
    result = value * value
END FUNCTION

DIM num AS N = 10
DIM numSquare AS N = 0
squareC(num, numSquare)
showvar("squareC result = " + numSquare)
```

An argument declared as `BYREF` is called "passed by reference." All function arguments, except object pointer variables, are passed by value (`BYVAL`) by default. When an argument is passed by value, a copy of the variable is made and sent to the function. If your variable contains a lot of data, such as the contents of a file, passing the variable by value can require a lot of memory. If a variable is passed by reference, however, the original variable is sent to the function and is not copied. Any modifications to a variable passed by reference changes the variable in the calling script.

Passing arguments by reference is often used by functions and methods in the Xbasic Function Library to return data from the function.

2.5.5 Function Pointers

Functions can be passed as arguments to other functions as function pointers. For example:

```
FUNCTION sayHello as V (name as C, formatter AS F)
    msg = "Hello " + name
    msg = formatter(msg)
    showvar(msg, "Hello")
END FUNCTION

sayHello("Jules",UPPER)
sayHello("Verne",LOWER)
```

The `sayHello()` function takes two arguments: `name` and `formatter`. `name` is a character variable added to the message, `msg`. `formatter` is a function pointer used to format the message. `formatter` can be any function that takes a character variable as an argument and returns a character value. The example demonstrates passing the `UPPER()` and `LOWER()` Xbasic functions when calling `sayHello()`. `UPPER()` converts a character string to upper case. `LOWER()` converts a character string to lower case.

2.6 Arrays, Object Pointer Variables, and Collections

In addition to simple data types – such as character, date, or time – Xbasic includes several built-in data structures for storing multiple related values together in a single variable. Arrays, collections (`U`), and object pointer variables (`P`) are data types that store multiple values in a single variable. These data structures include methods for doing things such as setting values, getting values, or retrieving information about the data structure.

2.6.1 Arrays

An array is a sequential series of data values of the same data type. Individual entries in an array are accessed using an index. Arrays are useful for collecting and processing sets of information, such as a set of scores or a list of names.

2.6.1.1 Declaring Arrays

An array can be declared as any data type (`A`, `B`, `C`, `D`, `K`, `L`, `N`, `P`, `T`, `U`, and `Y`.) Arrays are explicitly defined using the `DIM` statement with the bracket `[]` operator to declare the array's size. For example:

```
DIM names[5] as C
```

The example above declares a character array called `names` with a size of 5.

If the size of your array is unknown when you need to create it, you can declare it as a dynamic array. Dynamic arrays are declared by specifying zero (0) for the array size:

```
DIM locations[0] as C
```

Values can be assigned to an entry in an array using the bracket `[]` operator and a numeric value that indicates what entry in the array to update. In the example below, five names are assigned to the `names` array:

```
names[1] = "Amanda Higgins"
names[2] = "Nancy Clark"
names[3] = "Diane Morton"
names[4] = "John Rhodes"
names[5] = "Cecelia Dawkins"
```

Xbasic arrays are 1-indexed, meaning the first entry in the array is at index 1.

```
? names[1]
= "Amanda Higgins"

? names[5]
= "Cecelia Dawkins"
```

2.6.1.2 Declaring Arrays with Default Values

Arrays can be assigned a default value when they are declared using the assignment `=` operator. For example:

```
DIM str[4] AS C  = "Default"
? str
= [1] = "Default"
[2] = "Default"
[3] = "Default"
[4] = "Default"
```

The default value can be a literal (e.g., the string "Default" is a literal) or an expression. For example:

```
DIM nums[4] AS N = rand()
? nums
= [1] = 0.44000244140625
[2] = 0.837158203125
[3] = 0.128662109375
[4] = 0.797760009765625
```

In this example, the `nums` array is declared with a default value of `rand()`. The `rand()` function returns a random value between 0 and 1. The result is an array that contains 4 random numbers.

2.6.1.3 Adding New Entries to an Array

New entries can be added to an array using one of the following array methods: `push()`, `append()`, or `insert()`. These methods increase an array's size to make room for the new entry.

The `push()` method adds a value to the end of an array. For example:

```
locations.push("Ann Arbor,MI")
```

When the example executes, a new entry is created at the end of the `locations` array and set to the value "Ann Arbor, MI":

```
? locations
= [1] = "Ann Arbor,MI"
```

The `push()` method can be used with any array to add new entries. For example, `push()` can be used to add a new entry to the `names` array, which was created with a size of 5:

```
names.push("Matt Stevens")
```

After pushing a new value onto the end of the `names` array, the array now has six entries. EG:

```
? names[6]
= "Matt Stevens"
```

The array's `append()` method also adds new entries to the end of an array. `append()` creates a new blank entry at the end of the array and returns the index:

```
DIM newIndex AS N = names.append()
```

The index is used to set the value of the new entry:

```
names[newIndex] = "Lisa Hobbs"
```

If you need to add entries to the middle of an array, use the `insert()` method. The array `insert()` method adds one or more array entries at the specified index. For example, the code below inserts two new array entries in the `names` array at index:

```
names.insert(2,2)
```

After inserting new the array entries, you can assign a value to the new entries:

```
names[2] = "Peter Williams"
names[3] = "Anise Vanderbilt"
```

When the names array is output in the Interactive Window, it displays the list shown below:

```
? names
= [1] = "Amanda Higgins"
[2] = "Peter Williams"
[3] = "Anise Vanderbilt"
[4] = "Nancy Clark"
[5] = "Diane Morton"
[6] = "John Rhodes"
[7] = "Cecelia Dawkins"
[8] = "Matt Stevens"
```

2.6.1.4 Any Arrays

The data type contained in the array can be any valid variable data type, including the Any (`A`) data type. Assigning a value to an Any array element is similar to assign a value to a character or numeric array.

When a value is assigned to an Any array, however, the data is added as an object pointer with one property: `value`. `value` contains the value of the entry in the Any array. You must use the `value` property to read the value from the array. For example:

```
DIM arr[3] AS A
arr[1] = "char"
arr[2] = 5
arr[3] = now()

? arr[1]
= VALUE = "char"

? arr[1].value
= "char"
```

2.6.1.5 Multidimensional Arrays

Xbasic arrays can contain also contain multiple dimensions. For example, the following statement declares an array with 3 dimensions:

```
DIM a[1,1,3] as C
```

Assigning a value to a multidimensional array is done as follows:

```
a[1,1,1] = "First"
a[1,1,2] = "Second"
a[1,1,3] = "Third"

? a
= [1,1,1] = "First"
[1,1,2] = "Second"
[1,1,3] = "Third"
```

2.6.1.6 Looping Through Array Entries

The entries in an array are indexed sequentially starting at 1. You can loop or iterate over all the values of an array using a `FOR` loop to access each entry in the array. For example:

```
DIM values[10] AS N = rand()*10
DIM i AS N
DIM total AS N = 0
FOR i = 1 TO values.size()
    total = values[i] + total
NEXT i
showvar(total)
```

This example computes the total sum of all entries in the `values` array. The variable `i` is used to access the current element in the array in the `FOR` loop (`values[i]`) and add it to the `total`. The `size()`

method returns the size of the array. Using the `size()` method lets you create a loop that is independent of the size of the array, which is useful in cases where you don't know the size of the array.

The `size()` method takes an optional parameter that can be used to request the size for a specific dimension in a multidimensional array. For example:

```
DIM a[4,2] AS N
? a.size(1)
= 4

? a.size(2)
= 2
```

`size(1)` returns the size for the first dimension of the array, which is the value 4. `size(2)` returns the size for the second dimension of the array, which is the value 2.

The `size()` method can be used to create nested loops to iterate over every value in a multidimensional array. For example:

```
DIM i AS N
DIM j AS N
FOR i = 1 TO a.size(1)
    FOR j = 1 TO a.size(2)
        a[i,j] = j
    NEXT j
NEXT i

? a
= [1,1] = 1
[1,2] = 2
[2,1] = 1
[2,2] = 2
[3,1] = 1
[3,2] = 2
[4,1] = 1
[4,2] = 2
```

In this example, the multidimensional array `a` is populated by iterating over each entry in the array. The nested `FOR` loops are used to compute the index for the first and second dimension. `a[i,j] = j` assigns the value of `j` to the array entry located at `[i,j]`.

2.6.1.7 Array Methods

Xbasic arrays have multiple methods for working with arrays, including initializing array values, adding and removing entries, sorting, searching, exporting arrays to other formats, and getting information about the array (such as the array size.) We've shown how to use several array methods, including the `push()`, `append()`, `insert()`, and `size()` methods. In this section, we'll show how to perform some common tasks using arrays.

To learn more about the methods available for working with Xbasic arrays, see Array Methods[4] in the Alpha Anywhere documentation.

2.6.1.7.1 Initializing an Array using a Character List

An array can be populated with a character list using the array initialize()[5] method. For example:

```
DIM people AS C =<<%str%
Amanda Higgins
Nancy Clark
Diane Morton
John Rhodes
Cecelia Dawkins
%str%

DIM names2[0] AS C
names2.initialize(people)

? names2
= [1] = "Amanda Higgins"
[2] = "Nancy Clark"
[3] = "Diane Morton"
[4] = "John Rhodes"
[5] = "Cecelia Dawkins"
```

2.6.1.7.2 Merging two Arrays

You can combine multiple arrays of the same type into a single array using the append_arrays()[6] method. `append_arrays()` copies the entries from one or more arrays and adds them to the end of an array. For example, the Xbasic below merges the `trees` and `shrubs` arrays into a new array, `plants`:

[4] https://documentation.alphasoftware.com/index?search=api%20objects%20array%20methods
[5] https://documentation.alphasoftware.com/index?search=api%20objects%20array%20initialize%20function
[6] https://documentation.alphasoftware.com/index?search=api%20objects%20array%20append%20arrays%20function

```
DIM trees[0] AS C
trees.push("Oak")
trees.push("Pine")

DIM shrubs[0] AS C
shrubs.push("Lilac")
shrubs.push("Forsythia")

DIM plants[0] AS C
plants.append_arrays(trees,shrubs)
? plants
= [1] = "Oak"
[2] = "Pine"
[3] = "Lilac"
[4] = "Forsythia"
```

Arrays can only be merged if they are the same type and have one dimension. Attempting to merge two arrays of different types or merging multidimensional arrays will result in an error.

2.6.1.7.3 Deleting an Array Entry

Deleting array entries is done using the delete()[7] method. The `delete()` method takes two arguments: the index of the first element to delete and the number of elements to delete.

```
DIM fruit[5] AS C
fruit[1] = "Orange"
fruit[2] = "Banana"
fruit[3] = "Pear"
fruit[4] = "Apple"
fruit[5] = "Pineapple"
? fruit
= [1] = "Orange"
[2] = "Banana"
[3] = "Pear"
[4] = "Apple"
[5] = "Pineapple"

fruit.delete(2,2)  ' Delete "Banana" and "Pear"
? fruit
= [1] = "Orange"
[2] = "Apple"
[3] = "Pineapple"
[4] = ""
[5] = ""
```

If the array was declared with a fixed size, the empty array entries are moved to the end of the array, as shown in the example above. If the array was declared as a dynamic array, the entries are deleted and the array is resized:

[7] https://documentation.alphasoftware.com/index?search=api%20objects%20array%20delete%20function

```
DIM melons[0] AS C
melons.push("Honeydew")
melons.push("Cantaloupe")
melons.push("Watermelon")
melons.push("Casaba")
? melons
= [1] = "Honeydew"
[2] = "Cantaloupe"
[3] = "Watermelon"
[4] = "Casaba"

melons.delete(3) ' Delete "Watermelon"
? melons
= [1] = "Honeydew"
[2] = "Cantaloupe"
[3] = "Casaba"
```

2.6.1.7.4 Sorting an Array

Arrays can be sorted using the `sort()` method. Several optional parameters can be passed to the `sort()` method that define how to sort the array ("A" - ascending, "D" - descending).

```
dim numbers[10] as N = floor(rand()*100)+2
? numbers
= [1] = 2
[2] = 14
[3] = 67
[4] = 20
[5] = 83
[6] = 57
[7] = 56
[8] = 39
[9] = 40
[10] = 88

numbers.sort("A")
? numbers
= [1] = 2
[2] = 14
[3] = 20
[4] = 39
[5] = 40
[6] = 56
[7] = 57
[8] = 67
[9] = 83
[10] = 88
```

If the array is a property array, you can specify which property should be used to sort the array. For example, the Xbasic below sorts the `parents` array using the value in the `mom` property. The data is sorted alphabetically from A-Z (ascending):

```
DIM parents[2] AS P
parents[1].mom = "Irene"
parents[1].dad = "Abe"
parents[2].mom = "Arlene"
parents[2].dad = "Kyle"
parents.sort("A","mom")
```

See Array sort Method[8] in the online documentation to learn more about how to use the `sort()` method.

2.6.1.7.5 Searching an Array

The find()[9] and findi()[10] methods can be used to search an array. Both methods return the index of the location of the first element that matches the search expression. If no entries in the array match, the method returns 0. For example:

```
DIM squares[0] AS N
DIM num AS N

FOR num  = 1 TO 10
    squares.push(num*num)
NEXT num

? squares.find(36)
= 6

? squares[6]
= 36
```

The `find()` and `findi()` functions behave the same for all array types except character arrays. If the value to find is a character string, `find()` does a case-sensitive search while `findi()` does a case-insensitive search.

[8] https://documentation.alphasoftware.com/index?search=api%20objects%20array%20sort%20function
[9] https://documentation.alphasoftware.com/index?search=api%20objects%20array%20find%20function
[10] https://documentation.alphasoftware.com/index?search=api%20objects%20array%20findi%20function

```
DIM furniture[0] AS C
furniture.push("Chair")
furniture.push("Table")
furniture.push("Lamp")
furniture.push("Sofa")

? furniture.find("sofa")
= 0

? furniture.findi("sofa")
= 4

? furniture[4]
= "Sofa"
```

2.6.2 Object Pointer Variables

An object pointer, declared using the `P` type, is a special variable type in Xbasic that can contain multiple values defined as properties of the variable. Dot notation is used to define and access properties for object pointers. For example:

```
DIM person AS P
person.age = 47
person.firstName = "Liza"
person.lastName = "Stevens"
person.deceased = .F.
```

> ⚠ **Object Pointers are Dot Variables**
>
> Object pointers are also referred to as "dot variables" or "pointer variables." These terms are used interchangeably throughout the Alpha Anywhere documentation. For consistency, we call them object pointers in this guide.

You can use the `DIM` statement to add a property to an object pointer without assigning a value.

```
DIM person.address AS C
```

Properties can be any valid Xbasic variable type, including arrays, function pointers, and object pointers.

When passed to a function, object pointers are always passed by reference (`BYREF`). This means if the parameter is modified by the function, it changes the object pointer in the calling code. In the Interactive Window session shown below, the `lastname` property is added to the `person` variable by calling the `lastname()` function.

```
FUNCTION lastname AS V (person AS P)
    person.lastname = "Smith"
END FUNCTION

DIM p1 AS P
p1.firstname = "Janet"

? p1
= firstname = "Janet"

lastname(p1)

? p1
= firstname = "Janet"
lastname = "Smith"
```

2.6.2.1 Adding Methods to Object Pointer

Xbasic object pointers have methods that can be used to set and get values from the object. You can also add methods by adding function pointers as properties to the object pointer:

```
DIM obj AS P
obj.pet_name = "Savvy"
obj.breed = "Tuxedo"
obj.species = "Cat"

FUNCTION sayMeow AS V ()
    showvar("Meow!","Say 'Meow'")
END FUNCTION

DIM obj.speak AS F = sayMeow
obj.speak()
```

2.6.3 Object Pointer Arrays (Property Arrays)

A variable can be declared as an array of object pointers by declaring the array as type `P`. Referred to as "property arrays," each entry in a property array is an object pointer. For example:

```
DIM usPlaces[2] AS P
usPlaces[1].city = "Boston"
usPlaces[1].state = "MA"
usPlaces[2].city = "Atlanta"
usPlaces[2].state = "GA"
```

The code above defines a property array called `usPlaces` with a size of two. Each object pointer in the array is assigned a `city` and `state` property. The table below is a graphical representation of the property array defined above:

array element	city	state
usPlaces[1]	Boston	MA
usPlaces[2]	Atlanta	GA

2.6.3.1 Dynamic Property Arrays

New entries can be added to property arrays using the `append()` method. Properties can then be added to the new entry using the index returned by the `append()` method. For example:

```
dim pArr[0] as P
i = pArr.append()
pArr[i].name = "Fred"
pArr[i].city = "Boston"
pArr[i].age = 23

i = pArr.append()
pArr[i].name = "Tom"
pArr[i].city = "NY"
pArr[i].age = 35

? pArr.dump_properties("Name|city|age")
= Fred|Boston|23
Tom|NY|35
```

Property arrays also support special bracket syntax for adding new entries to the array, which you may encounter in Xbasic found in the Alpha Anywhere documentation or user forum. The syntax uses the `[]` and `[..]` array operators. The `[]` operator creates a new element at the end of the array, and the `[..]` operator adds properties to the last element in the array.

```
dim pArr2[1] as P
pArr2[1].name = "Fred"
pArr2[1].city = "Boston"
pArr2[1].age = 23

pArr2[].name = "Tom"
pArr2[..].city = "NY"
pArr2[..].age = 35

? pArr2.dump_properties("Name|city|age")
= Fred|Boston|23
Tom|NY|35
```

In the code shown above, `a[].name = "Tom"` adds array element number 2. `a[..]. city = "NY"` adds "NY" to the newly created array element (i.e. number 2).

While this syntax is supported, it is not a recommended best practice to use `[]` and `[..]` to dynamically append elements to a property array. If you forget to use `[..]` to add properties and instead use `[],` you can end up in the following scenario:

```
dim pArr3[0] as p
pArr3[].name = "Fred"
pArr3[].city = "Boston"
pArr3[].age = 23
pArr3[].name = "Tom"
pArr3[].city = "NY"
pArr3[].age = 35

? pArr3.dump_properties("Name|city|age")
= Fred||
|Boston|
||23
Tom||
|NY|
||35
```

Instead of creating two array entries with a name, city, and age property, the script creates an array that contains six entries with a name, city, or age property. To avoid this issue, use the `append()` method.

2.6.3.2 Using Property Arrays with JSON Data

Object pointer arrays can be used with JSON data in Xbasic. JSON is used on the client in UX controls, such as the List and ViewBox, as well as by web services, including the Alpha TransForm API. When working with JSON data in Xbasic, it is often easier to convert the JSON into an object pointer to work with the data.

```
DIM json AS C =<<%json%
[
    {"fname":"Alicia","lname":"Davis","city":"Springfield","state":"VA"},
    {"fname":"Reuben","lname":"Hayes","city":"Anchorage","state":"AK"},
    {"fname":"Joel","lname":"Kay","city":"Boise","state":"ID"}
]
%json%

DIM people AS P
people = json_parse(json)
```

When you're done manipulating the data, it can be serialized back into JSON before sending it to the client.

```
json = json_generate(people)
```

2.6.4 Built-in Xbasic Objects

Xbasic Objects are global variables available to your scripts. Built-in Xbasic objects are similar to object pointer variables in that they have properties. They also often contain methods -- functions you can call. Two objects frequently used in web applications are the `context` and `TRACE` objects.

The `context` object contains the Response, Request, Security, and Session objects. These four objects store state information for web applications and provide methods to read and write cookies, handle web page redirects, work with the security framework, and store data in the user's session.

- `context.request` - This object represents the parsed HTTP request that was received by the Application Server and has several properties and methods.
- `context.response` - This object represents the HTTP response created by the Application Server and sent back to the client. It has several properties and methods. The server creates this object for each Response and can be accessed directly by your applications.
- `context.session` – This object contains data about an individual user's session, including session variables and temporary session files.
- `context.security` – This object contains properties and methods available for working with the users and roles if a web application uses security.

The `TRACE` object is useful for logging messages in the server logs. It can be used in both the development environment and published applications. Messages written to the `TRACE` logs can be accessed either in the Trace Window (Developer IDE) or the Trace Folder on the Application Server. `TRACE` logs on Alpha Cloud can also be accessed through the Alpha Cloud interface in the Developer IDE.

Another object you may also use is the `FILE` object. `FILE` is used for working with files stored on the server running Alpha Anywhere. `FILE` is not supported on Alpha Cloud.

You can find information about available objects online in the Alpha Anywhere Documentation. See Xbasic Objects[11].

[11] https://documentation.alphasoftware.com/index?search=xbasic%20objects

2.6.5 Collections

A collection is an Xbasic datatype, similar to an array, but whose elements are referenced with a key rather than an index. A collection is declared using the data type `U`:

```
DIM collection AS U
```

Elements are added to a collection using the `set()` method. The `set()` method takes two parameters: a key and a value. The key is a character, numeric, or date value that uniquely identifies an element in a collection. The value is the data stored in the collection for the specified key and can be any value or expression that evaluates to a single value. For example:

```
DIM people AS U
people.set("FJ","Fred Jones")
people.set("BB","Bryan Boyd")
people.set("KL","Kim Lee")
```

In the code example above, a collection is created named `people,` and three entries are added to the collection using the `set()` method.

To get the value in a collection, you can use the `get()` method. The `get()` method takes a key and returns the value. For example, the Xbasic below retrieves and prints the value for the key "BB" for the `people` collection:

```
? people.get("BB")
= "Bryan Boyd"
```

By contrast, in an array, elements are referred to by index number. In the array shown below, the second element corresponds to the value "Bryan Boyd."

```
DIM persons[0] AS C
persons.push("Fred Jones")
persons.push("Bryan Boyd")
persons.push("Kim Lee")

? persons[2]
= "Bryan Boyd"
```

See "Collection Methods[12]" in the online documentation to learn more about collections.

[12] https://documentation.alphasoftware.com/index?search=api%20collection%20object%20collection%20function

2.7 Capturing and Logging Errors

If an unexpected problem occurs when a script executes (i.e., a runtime error), Xbasic generates an error message. Unhandled errors may be shown to the user if they occur in an Ajax callback or server-side event. Because Xbasic runtime errors are often cryptic and cannot be fixed by the end-user, you want to include error handling to trap and log errors when they happen.

2.7.1 ON ERROR GOTO

An easy way to avoid sending cryptic error messages to your users is to use an error handler. The `ON ERROR...GOTO` statement can be wrapped around any Xbasic code to add error handling. For example:

```
FUNCTION myFunction AS C ()

ON ERROR GOTO HandleError

    'Xbasic Code to Execute

    'If we reach this point, no errors occurred!
    'Exit the function to prevent executing the HandleError code
    EXIT FUNCTION

HandleError:
    'Xbasic to handle the error
    'Get the error code:
    err = error_code_get()
    'Get the error message:
    msg = error_text_get(err)

    'Log error message to TRACE log on server:
    log_msg = "Encountered Error #" + err + " in myFunction(): " + msg
    TRACE.writeLn(log_msg,"myFunction")

    myFunction = "Problem executing callback."

END FUNCTION
```

The `ON ERROR...GOTO` statement defines a specific location to jump during code execution when a runtime error occurs. The location to go to is a label, specified using the syntax

```
labelname:
```

A label defines a place that can be jumped to using the `GOTO` statement. The label and error handling code should be defined at the end of your script. The label to go to is specified at the end of the `ON ERROR...GOTO` line at the beginning of your script:

```
ON ERROR GOTO labelname
```

Unlike `IF` statements, when the label is encountered after an `ON ERROR...GOTO` declaration, it is not skipped. A label is simply a named location that can be jumped to using the `GOTO` statement. In order to prevent the execution of error-handling code specified after the label, you must use an `END`, `EXIT FUNCTION`, or `RETURN` statement before the label is encountered.

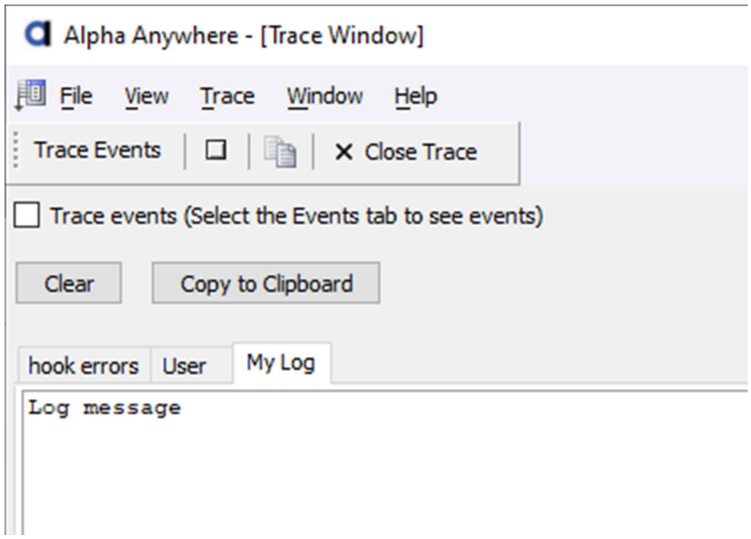
2.7.2 Logging: Using the Trace Log

Log files are useful for recording information about events in a published application. If an error occurs in your application, you can use a log file to capture the error message and other details that you need to know to diagnose and correct the issue. The Trace Log is an ideal place to write out error information. The Trace Log is available both in the IDE (the Trace window) and in a deployed application (the Trace Log folder).

To send text output to the Trace Log, you can use the `TRACE.writeLn()` method. The `TRACE.writeLn()` method takes two parameters: the message to write to the Trace Log and an optional log name. For example, run the following code in the Interactive Window:

```
TRACE.writeLn("Log message","My Log")
```

To view the Trace Log, select View > Trace Window from the main menu on the Web Projects Control Panel. Locate the "My Log" tab in the Trace Window and open it. You should see the message "Log message" in the message window:



If you do not specify a log name for the second parameter, the message is written to the "User" log.

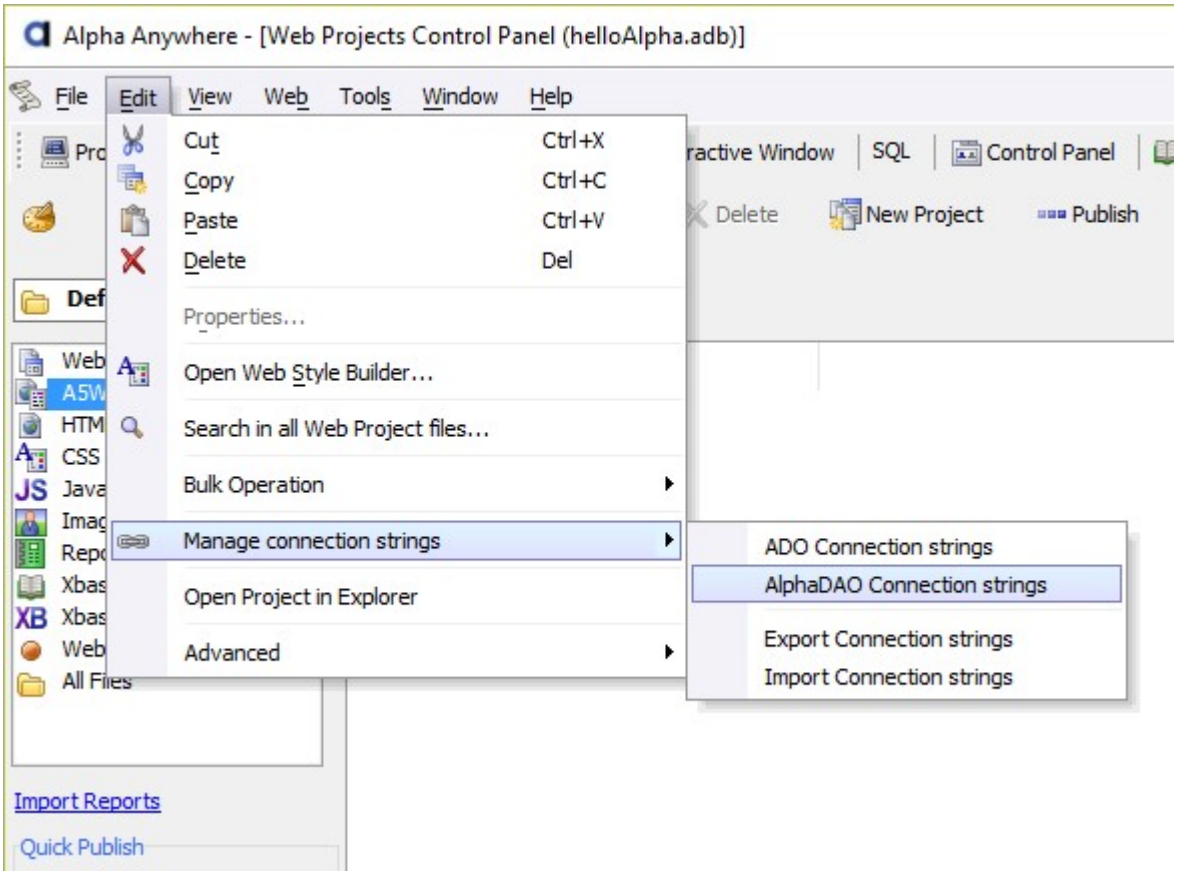
3 Working with SQL Data Using Xbasic

Most applications build in Alpha Anywhere communicate with a database -- such as MySQL or SQL Server. Xbasic provides powerful commands for working with data in a SQL database. Understanding how to use these commands enables you to build complex workflows in your applications beyond what Alpha Anywhere provides out of the box.

3.1 AlphaDAO Connections

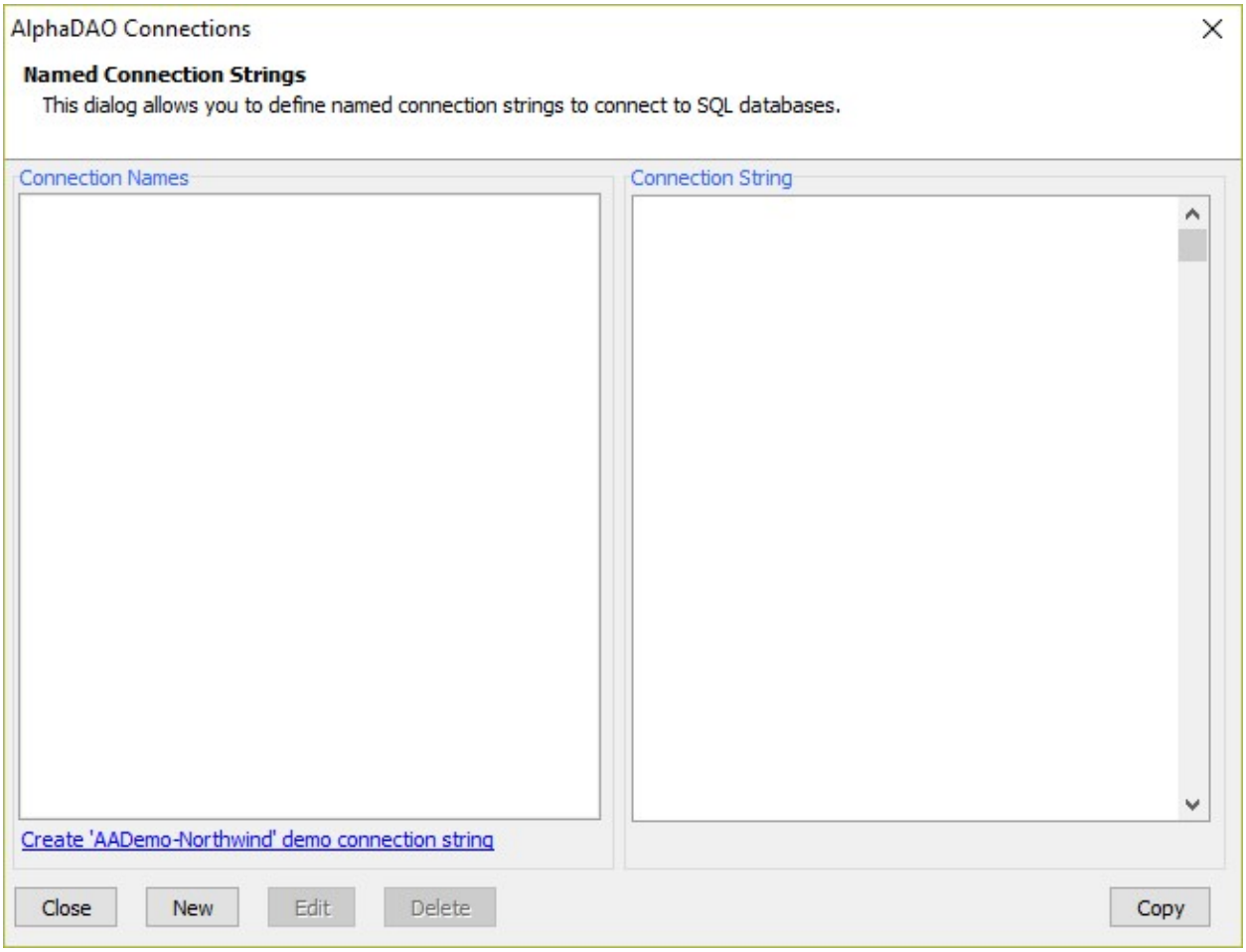
Alpha Anywhere communicates with a database using an AlphaDAO connection string. AlphaDAO stands for "Alpha Anywhere Data Access Object". AlphaDAO is an interface through which you access data stored in SQL, NoSQL, DBaaS, SaaS, and other data sources, including static JSON and OData (Open Data Protocol) APIs. There are several methods for creating a connection string, including Ad-hoc and Named Connections.

AlphaDAO connections can be created using the AlphaDAO Connections dialog, found under the Edit menu on the Web Projects Control Panel.

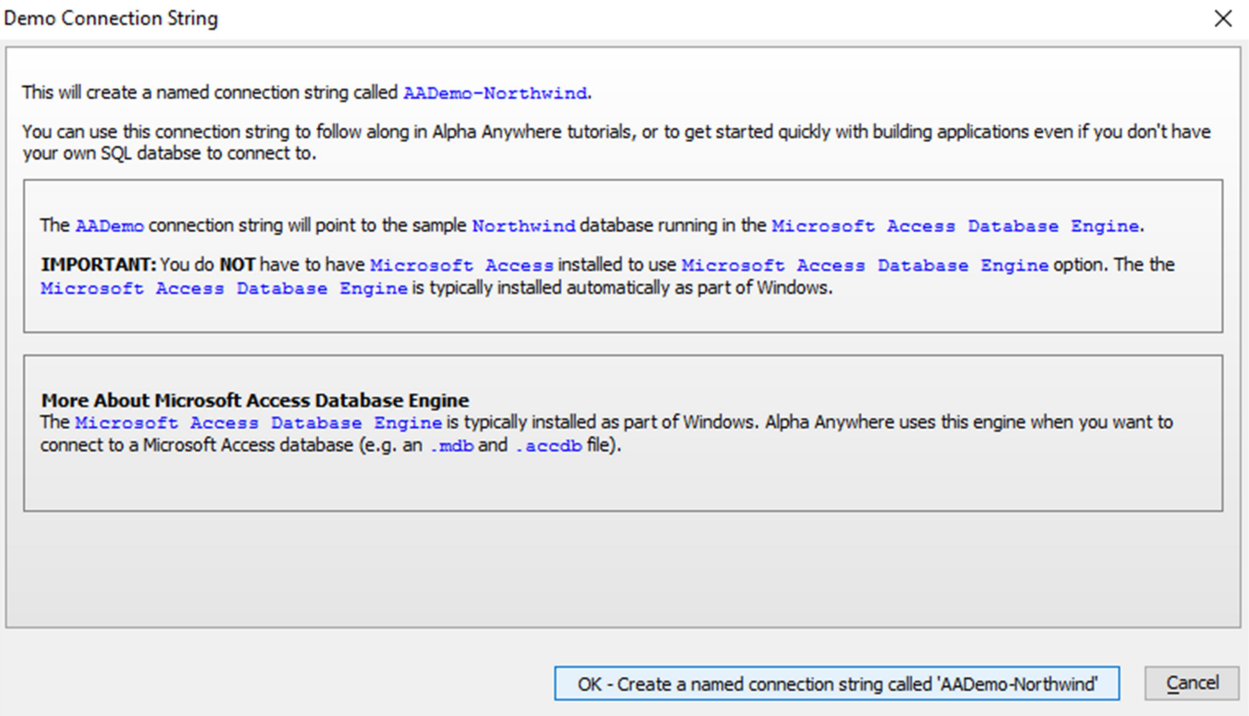


Named connection strings are created and managed using the AlphaDAO connections dialog. For the SQL examples in this section, we will use the Microsoft Access Northwind database. Alpha Anywhere includes a pre-defined connection string, "AADemo-Northwind", for communicating with the Northwind

database. This connection string can be added to your workspace by clicking the "Create 'AADemo-Northwind' demo connection string" link at the bottom of the AlphaDAO Connections dialog.



Follow the instructions to create the connection string and close the AlphaDAO Connections dialog after the AADemo-Northwind named connection has been created.



3.2 The SQL Namespace

The SQL namespace is a collection of Xbasic classes used for communicating with and performing operations on a database or any system of record that you can connect to with an AlphaDAO connection. The core SQL namespace classes used to interact with a SQL database in an Xbasic script are:

- `SQL::Connection` - The `SQL::Connection` object is used to open a connection to a SQL database and execute commands against the database. Commands are specified using SQL – Structured Query Language.
- `SQL::ResultSet` – The `SQL::ResultSet` object contains the data retrieved from a database after executing a SQL SELECT command.
- `SQL::Arguments` – The `SQL::Arguments` object is used to pass values to a SQL command. An argument is a variable used in an SQL command, such as in WHERE or ORDER BY clauses, in place of constant values. Always use arguments when building SQL commands.
- `SQL::CallResult` – The `SQL::CallResult` object contains detailed information about the success of calling a method of a `SQL::Connection` or `SQL::ResultSet` object. If executing a SQL command fails, the `CallResult` object contains additional information about why the command failed.

Using these four classes, you can perform CRUD (create, read, update, delete) operations against a SQL database directly from Xbasic.

3.3 Connecting to the Database

Before you can query the SQL database, you must first open a connection to the database. Connections are opened using the `SQL::Connection` `open()` method:

```
DIM conn AS SQL::Connection
DIM connStr AS C = "::Name::AADemo-Northwind"

success = conn.open("::Name::AADemo-Northwind")
```

The `open()` method takes a named or ad-hoc AlphaDAO connection string and opens a communications channel to the database. The example above uses the AADemo-Northwind named connection (see "AlphaDAO Connections" on page 53 if you do not have the AADemo-Northwind named connection.) If the connection is established successfully, `open()` returns `.T.`. If the connection fails, `open()` returns `.F.`.

If the connection fails, the `SQL::Connection`'s `CallResult` object contains additional details as to why the call failed:

```
DIM cr AS SQL::CallResult
cr = conn.callResult

IF (.not. success) THEN  'Could not open connection
    'Get the error message from the SQL::CallResult
    DIM errorMsg AS C
    errorMsg = cr.text

    'Write the message to the Trace log
    TRACE.writeLn(errorMsg,"SQL Error")

    'Terminate the script
    END
END IF
```

You should always check the return value of the `open()` method before attempting to perform a query against a database. It is also a best practice to copy the `SQL::Connection` `CallResult` property into a `SQL::CallResult` variable to ensure the error message is preserved when working with SQL functions.

3.4 Executing a Query

Once a connection has been established, you can execute queries against the database. The `SQL::Connection` `execute()` method can be used to perform any CRUD operations against a SQL database.

```
DIM sqlQuery AS C = "SELECT * FROM Customers"
success = conn.execute(sqlQuery)
cr = conn.callResult
```

The `execute()` method also returns a `.T.` or `.F.` value indicating whether or not the SQL query succeeded. Most methods of the `SQL::Connection` object return a value indicating their success. If the method call fails, the `CallResult` object contains additional information about the operation's failure.

```
IF (success) THEN
    'Process the Query results
ELSE
    'SQL statement failed to execute:
    DIM errorMsg AS C
    errorMsg = cr.text

    'Write the message to the Trace log
    TRACE.writeLn(errorMsg,"SQL Error")
END IF
```

3.5 Processing the Query Results

If a SQL query returns any records, the `SQL::Connection`'s `ResultSet` property will contain one or more rows of data. You can use the `nextRow()` method to access each record returned by the query. The `nextRow()` method steps through each record returned by the query. The first time you access records in a `SQL::ResultSet`, the current record pointer is positioned before the first row. Calling `nextRow()` advances the record pointer to the first record. Subsequent calls to `nextRow()` advance the record pointer to the next record. When there are no more available records, `nextRow()` returns `.F.`.

```
WHILE conn.resultSet.nextRow()
    'Code to process the current row
END WHILE
```

The `SQL::ResultSet` is a "forwards only" object. Records are processed from first to last; you cannot access previously seen records in a `SQL::ResultSet`. You either need to execute the query a second time or store the records from the `SQL::ResultSet` in a variable.

3.5.1 Reading Data from the Current Record

Specific field (or column) values in a `SQL::ResultSet` are accessed using the `data()` method. The `data()` method returns the value for a column in the current row. The column is specified either as the column name or the column's number. For example, in the script below, the `data()` method is used to get the value of the "country" field:

```
DIM countries AS C
DIM country AS C
WHILE conn.resultSet.nextRow()
    'Get the country from the current record
    country = conn.resultSet.data("country")

    'Test to see if the country is in the list
    IF (country !$ countries) THEN
        'country was not found in the list
        'Add the country to the countries list
        countries = countries + country + crlf()
    END IF
END WHILE
```

If you know the order of the columns in the query, you can use the column's number instead of the column name. For example:

```
DIM sqlSelect AS C =<<%sql%
SELECT city, country FROM Customers
WHERE CustomerId = :CustomerID
%sql%

DIM args as SQL::Arguments
args.set("CustomerID","BOLID")

DIM country AS C = ""
IF (conn.execute(sqlSelect, args) <> .F.) THEN
    IF (conn.resultSet.nextRow() <> .F.) THEN
        country = conn.resultSet.data(2)
    END IF
END IF

? country
= "Spain"
```

The SQL `SELECT` statement specifies 2 columns in the query: `city` and `country`. `city` is the first column, and `country` is the second column. `conn.resultSet.data(2)` returns the value of the `country` column for the current row of data in the result set.

Specifying the column number can be faster in some situations. Using the column number also allows for dynamically retrieving data from a result set row using a loop. Use the `conn.resultSet.ColumnCount` property to determine the total number of columns in the result set.

3.6 Closing Connections

When you are done querying the database, you should close the connection. Connections are closed using the `SQL::Connection` `close()` method.

```
conn.close() 'Close the connection
```

3.7 Creating Queries with Arguments

Most SQL queries include `WHERE` clauses to filter the results. For example, you may only want to fetch a list of customers from Spain. The SQL query to do this may look like this:

```
DIM sqlSelect AS C = "SELECT * FROM Customers WHERE Country = 'Spain'"
```

If you wanted to give the user a choice as to what country to get customer data for, however, you will need to define the SQL `WHERE` clause using arguments in place of static values.

Arguments allow you to define a SQL statement where parts of the statement are determined dynamically. For example, the above `SELECT` statement can be rewritten using SQL arguments as follows:

```
country = "Spain"
DIM sqlSelect AS C = "SELECT * FROM Customers Where Country = :Country"

DIM args AS SQL::Arguments
args.set("Country",country)
```

The country variable represents the data we received from the user. In a web application, this data may be passed to the Xbasic script via an Ajax callback or session variable.

```
'Read the country from the data submitted
'to this Ajax callback function:
country = e.dataSubmitted.selectedCountry
```

You might be wondering why we did not use string concatenation to create the query. If the query is populated with data gathered from the user, you risk exposing your database to common SQL vulnerabilities if you do not pre-process the data submitted by the user before using it in a query. Values set in `SQL::Arguments` are sanitized before being used in SQL statements, protecting you from SQL Injection attacks and other common SQL hacks. Because of this, **you should always use arguments in SQL queries.**

3.8 Converting Query Results to Other Formats

The `ResultSet` includes methods for converting the query results to another data format, including character lists, property arrays, JSON, XML, CSV, and Excel.

3.8.1 Converting a ResultSet to an Xbasic Variable

It's sometimes easier to deal with a result set by saving the data into an Xbasic Variable, such as a character list or property array. The `SQL::ResultSet` has several methods for converting a result set to a variable: `toString()` and `toPropertyArray()`.

The `toString()` method formats the results set as a character list. The first line in the character list contains the column names. The lines that follow are the records returned by the SQL query. For example, run the following code in the Interactive Window:

```
DIM conn AS SQL::Connection
DIM cr AS SQL::CallResult
DIM args AS SQL::Arguments

DIM sqlQuery AS C = "SELECT * FROM Customers WHERE Country = :Country"
args.set("Country","Spain")

DIM recordList AS C
IF (conn.open("::Name::AADemo-Northwind")) THEN
    IF (conn.execute(sqlQuery,args)) THEN
        recordList = conn.resultSet.toString()
    ELSE
        cr = conn.callResult
        TRACE.writeLn(cr.text,"SQL Error")
        conn.close()
        END
    END IF
    conn.close()
ELSE
    cr = conn.callResult
    TRACE.writeLn(cr.text,"SQL Error")
    END
END IF

'Reformat UTF8 database data to ACP, which is used by the Interactive Window
recordList = convert_utf8_to_acp(recordList)

? recordList
```

The `toPropertyArray()` method is similar to `toString()` in that it converts the result set to a format stored in an Xbasic variable. Instead of formatting the data as a CR-LF delimited list of strings, however, `toPropertyArray()` converts the result set to a property array where each record is an entry in the array, and each field is a property of an array entry.

```
DIM conn AS SQL::Connection
DIM cr AS SQL::CallResult
DIM args AS SQL::Arguments

DIM sqlQuery AS C = "SELECT * FROM Customers WHERE Country = :Country"
args.set("Country","Spain")

DIM recordArr[0] AS P
IF (conn.open("::Name::AADemo-Northwind")) THEN
    IF (conn.execute(sqlQuery,args)) THEN
        conn.resultSet.toPropertyArray(recordArr)
    ELSE
        cr = conn.callResult
        TRACE.writeLn(cr.text,"SQL Error")
        conn.close()
        END
    END IF
    conn.close()
ELSE
    cr = conn.callResult
    TRACE.writeLn(cr.text,"SQL Error")
    END
END IF

'Output the number of records
? recordArr.size()

'Output the first record
? recordArr[1]

'Output the value of customerId in first record
? recordArr[1].customerId
```

3.8.2 Converting a ResultSet to JSON, XML, or CSV

In addition to converting a result set to a variable, you can also convert the result set to another data format, such as JSON. JSON is frequently used when processing SQL data to display in a List control. For example:

```
FUNCTION getCustomerData AS C ()
    DIM conn AS SQL::Connection
    DIM cr AS SQL::CallResult
    DIM sqlQuery AS C = "SELECT * FROM Customers"

    DIM recordJSON AS C
    IF (conn.open("::Name::AADemo-Northwind")) THEN
        IF (conn.execute(sqlQuery)) THEN
            recordJSON = conn.resultSet.toJSON()
        ELSE
            cr = conn.callResult
            TRACE.writeLn(cr.text,"SQL Error")
            conn.close()
            END
        END IF
        conn.close()
    ELSE
        cr = conn.callResult
        TRACE.writeLn(cr.text,"SQL Error")
        END
    END IF

    RETURN recordJSON
END FUNCTION
```

The result set can be converted to other formats as well, including XML (the `toXML()` method) and Comma Separated Variable format (the `toCSV()` method).

3.8.3 Writing a ResultSet to a JSON or Excel File

A result set can be written out to file in a variety of formats. This includes JSON, XML, CSV, and Excel. For example, the script below writes the customer table out to an Excel file named "Customers.xlsx":

```
DIM conn AS SQL::Connection
DIM cr AS SQL::CallResult
DIM args AS SQL::Arguments

DIM sqlQuery AS C = "SELECT * FROM Customers WHERE Country = :Country"
args.set("Country","Spain")

DIM recordList AS C
IF (conn.open("::Name::AADemo-Northwind")) THEN
    IF (conn.execute(sqlQuery,args)) THEN
        conn.resultSet.toExcel("C:/spreadsheets/Customers.xlsx")
    ELSE
        cr = conn.callResult
        TRACE.writeLn(cr.text,"SQL Error")
        conn.close()
        END
    END IF
    conn.close()
ELSE
    cr = conn.callResult
    TRACE.writeLn(cr.text,"SQL Error")
    END
END IF
```

Other functions available for writing the result set to file include `toCSVFile()`, which writes the result set to a file in Comma Separated Variable format, and `toJSONFile()`, which converts the result set to JSON format and saves it to a file.

3.9 Transactions

Many database systems allow you to perform updates to tables within the context of a transaction. Transactions are useful when you want to make multiple updates to one or more tables if and only if all of the updates are successful. Statements executed during a transaction are applied (e.g., committed) when the transaction is committed. If something happens that requires undoing (e.g., rolling back) all of the changes to the database, however, the transaction can be rolled back instead.

The `SQL::Connection` object provides the following methods for wrapping your queries inside a transaction:

- `beginTransaction()` -- Starts a transaction. All queries executed after calling this method are included in the transaction.
- `rollbackTransaction()` -- Reverts all changes to the database made during the transaction, returning the database to the state it was before executing any statements and ends the transaction.
- `commitTransaction()` -- Commits the changes to the database and ends the transaction.

The workflow for transactional queries is shown on the next page:

```
DIM conn AS SQL::Connection
DIM cr as SQL::CallResult
IF (conn.open("::Name::AADemo-Northwind") THEN

    'Perform any queries that don't need to be transacted here

    'Begin the transaction
    conn.beginTransaction()

    'Execute the query or queries that need to be transacted here
    '(Note: Replace sqlUpdateQueries with your SQL queries)
    success = conn.execute(sqlUpdateQueries)

    'Capture the CallResult
    cr = conn.callResult

    'Validate the query or queries succeeded
    IF (success) THEN
        'Query or queries succeeded; commit transaction
        conn.commitTransaction()
    ELSE
        'Query or queries failed; rollback transaction
        conn.rollbackTransaction()

        'Optional, but recommended: log any error messages
        TRACE.writeLn(cr.text,"SQL Error")
    END IF
END IF

conn.close() 'Close the connection
```

If creating the transaction fails or the database does not support transactions, the `beginTransaction()`, `commitTransaction()`, and `rollbackTransaction()` statements will return `.F.`. The `SQL::Connection`'s call result object will contain any additional details as to why the method(s) failed.

While using transactions is good practice when executing a batch of `INSERT`, `UPDATE`, or `DELETE` queries, not all queries can be transacted. Some queries are permanent once they have been executed and cannot be undone using transactions. For example, MySQL does not support making database and table structure changes (e.g., `ALTER TABLE`) in a transaction. Altering or dropping a table is a permanent action in MySQL and cannot be undone using transactions.

You can also commit your transactions prematurely if you execute a statement that performs an implicit commit. Administrative queries (such as getting the table info using the `SQL::Connection getTableInfo()` method) or nesting transactions may automatically commit the current transaction when they execute.

In general, you should keep your transaction as short as possible. This can be accomplished by gathering all of the data you need upfront and only transaction queries that require it. For other types of queries

that are not `INSERT`, `UPDATE`, or `DELETE` statements, you should consult your database documentation to ensure they are supported in transactions and do not cause any unwanted side effects.

3.10 Writing Portable SQL Queries

SQL (Structured Query Language) is not a standard syntax. While most database management systems use SQL to interact with the database, each SQL database vendor provides an implementation of SQL that is not compatible with other database systems. For example, the three queries shown below fetch the same data from the Northwind `Customers` table stored in different database systems:

```
' Access:
sql = "SELECT CompanyName, City & Region, Time() FROM Customers"

' MySQL:
sql = "SELECT CompanyName, City + Region, CURRENT_TIMESTAMP FROM Customers"

' SQL Server:
sql = "SELECT CompanyName, Concat(City, Region), CurTime() FROM Customers"
```

If you only work with one database system, the lack of a standard syntax for SQL may not pose any issues. However, if you develop Software-as-a-Service (SaaS) systems, work with multiple database systems, or may migrate your systems of record to a different vendor in the future, using native SQL syntax can be problematic.

To solve the problem of the lack of SQL standardization and ensure that applications built with Alpha Anywhere can seamlessly integrate with any database back-end, Alpha Anywhere supports Portable SQL. Portable SQL is a database-independent standardized SQL syntax with built-in functions. The three database-specific SQL queries shown above can be re-written using Portable SQL as follows:

```
SELECT CompanyName, Concatenate(City, Region), CurrentTime() FROM Customers
```

Alpha Anywhere automatically translates portable SQL to the native SQL syntax used by the target database system at run-time. Queries are written once using portable SQL and executed on any database system supported in Alpha Anywhere.

The `SQL::Connection` object defines whether or not the portable SQL parser is used when processing a query. By default, when a `SQL::Connection` object is created, portable SQL is disabled. To enable portable SQL, set the `portableSQLEnabled` property of the `SQL::Connection` object to true (`.T.`):

```
DIM conn AS SQL::Connection
? conn.portableSQLEnabled
= .F.

conn.portableSQLEnabled = .T.
? conn.portableSQLEnabled
= .T.
```

Portable SQL can be used to create any SQL `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement.

3.10.1 Portable INSERT Statements

The portable SQL `INSERT` statement adds one or more records to a table. The basic format of an `INSERT` statement is

```
INSERT INTO tableName (colName1, colName2, ..., colNameN)
VALUES (colVal1, colVal2, ..., colValN)
```

`colName1` through `colNameN` correspond to the column names in the table, while `colVal1` through `colValN` are the values to set in those columns. The number of columns specified must match the number of values. Values can either be literal values or an expression.

`SQL::Arguments` should always be used with `INSERT` statements. For example:

```
DIM conn AS SQL::Connection
DIM cr AS SQL::CallResult
DIM sqlInsert AS C =<<%sql%
INSERT INTO Customers (customerid,companyname,city,country)
VALUES (:customerid,:companyname,:city,:country)
%sql%
DIM args AS SQL::Arguments
args.set("customerid","SAMPL")
args.set("companyname","Sample Company")
args.set("city","Lewiston")
args.set("country","USA")

IF (conn.open("::Name::AADemo-Northwind")) THEN
    IF (conn.execute(sqlInsert,args)) THEN
        ' Insert successful; log a success message to the SQL Log.
        DIM affectedRows AS N = conn.affectedRows()
        DIM msg AS C
        msg = "Insert Successful! " + affectedRows + " added."
        TRACE.writeLn(msg,"SQL Log")
    ELSE
        ' Insert failed; log a failure message to the SQL Log.
        cr = conn.callResult
        TRACE.writeLn(cr.text,"SQL Log")
    END IF
ELSE
    ' Connection failed: log a failure message to the SQL Log.
    cr = conn.callResult
    TRACE.writeLn(cr.text,"SQL Log")
END IF
conn.close()
```

3.10.2 Portable UPDATE and DELETE Statements

SQL `UPDATE` and `DELETE` statements also modify the data in a database. However, unlike the `INSERT` statement, they modify existing records. It is crucially important that you always include a `WHERE` clause in your `UPDATE` and `DELETE` statements. Without a `WHERE` clause, you risk modifying (or deleting) all records in a table.

`SQL::Arguments` should always be used with `UPDATE` and `DELETE` statements.

The syntax of a portable SQL `UPDATE` statement is shown below:

```
UPDATE tablename
SET
field1 = value1,
field2 = value2,
fieldN = valueN
WHERE where_clause
```

For example:

```
DIM conn AS SQL::Connection
DIM cr AS SQL::CallResult
DIM sqlUpdate AS C =<<%sql%
UPDATE Customers
SET
city = :city,
region = :region
WHERE customerid = :customerid
%sql%

DIM args AS SQL::Arguments
args.set("customerid","SAMPL")
args.set("city","Montpelier")
args.set("region","VA")

IF (conn.open("::Name::AADemo-Northwind")) THEN
    IF (conn.execute(sqlUpdate,args)) THEN
        ' Update successful; log a success message to the SQL Log.
        DIM affectedRows AS N = conn.affectedRows()
        DIM msg AS C
        msg = "Insert Successful! " + affectedRows + " updated."
        TRACE.writeLn(msg,"SQL Log")
    ELSE
        ' Insert failed; log a failure message to the SQL Log.
        cr = conn.callResult
        TRACE.writeLn(cr.text,"SQL Log")
    END IF
ELSE
    ' Connection failed: log a failure message to the SQL Log.
    cr = conn.callResult
    TRACE.writeLn(cr.text,"SQL Log")
END IF
conn.close()
```

The syntax of a portable SQL `DELETE` statement is shown below:

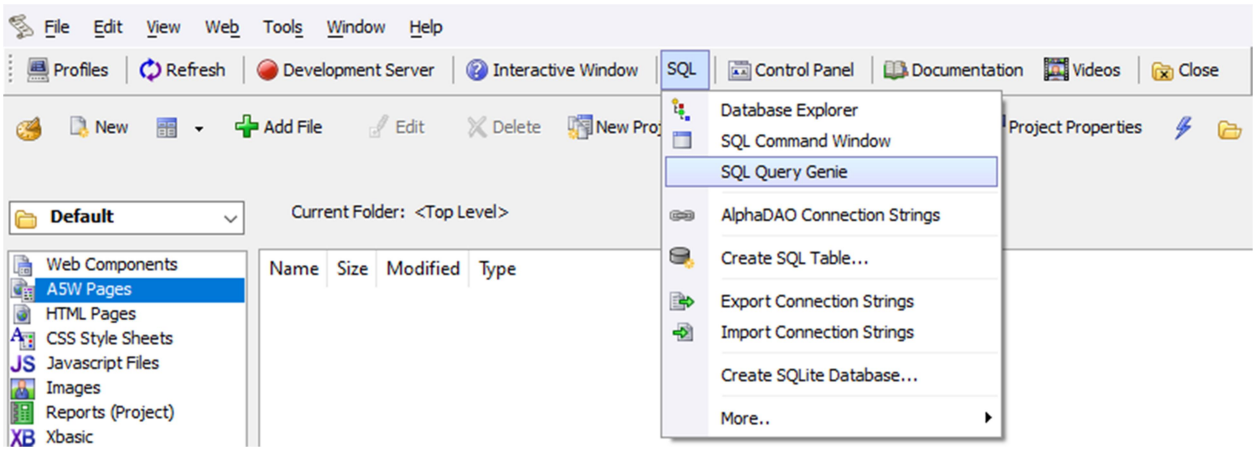
```
DELETE FROM tablename
WHERE where_clause
```

For example:

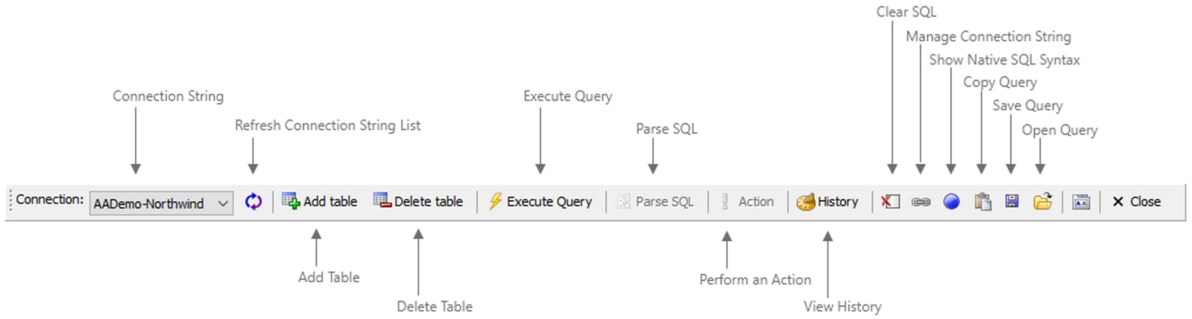
```
DIM conn AS SQL::Connection
DIM cr AS SQL::CallResult

dim sqlDelete AS C =<<%sql%
DELETE FROM customers
WHERE customerid = :customerid
%sql%

DIM args AS SQL::Arguments
args.set("customerid","SAMPL")

IF (conn.open("::Name::AADemo-Northwind")) THEN
    IF (conn.execute(sqlDelete,args)) THEN
        ' Delete successful; log a success message to the SQL Log.
        DIM affectedRows AS N = conn.affectedRows()
        DIM msg AS C
        msg = "Delete Successful! " + affectedRows + " deleted."
        TRACE.writeLn(msg,"SQL Log")
    ELSE
        ' Delete failed; log a failure message to the SQL Log.
        cr = conn.callResult
        TRACE.writeLn(cr.text,"SQL Log")
    END IF
ELSE
    ' Connection failed: log a failure message to the SQL Log.
    cr = conn.callResult
    TRACE.writeLn(cr.text,"SQL Log")
END IF
conn.close()
```

3.10.3 SQL Query Genie

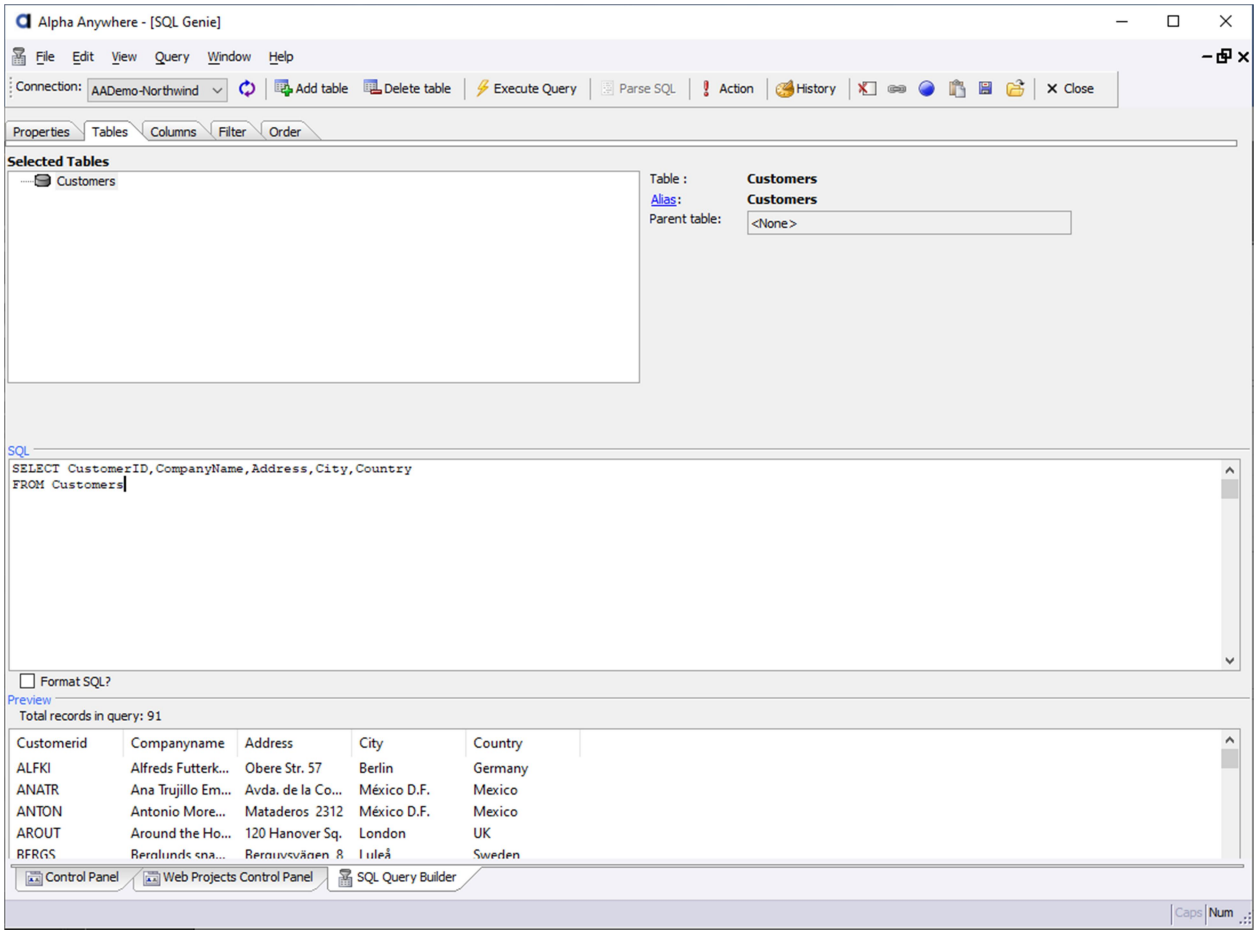
The SQL Query Genie is a handy tool for building SQL select statements. The genie is located under the SQL menu on the Web Projects Control Panel.



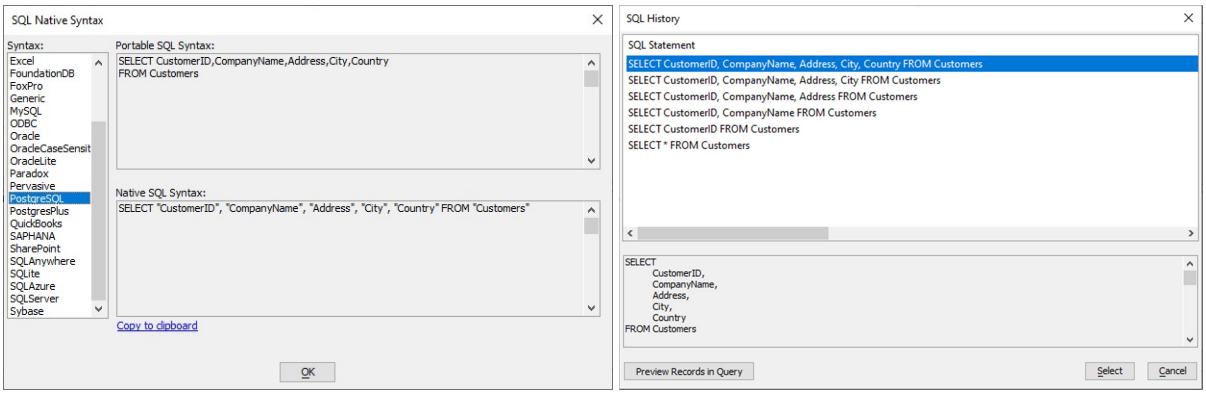
Using this genie, you can create and test both native and portable SQL queries for selecting or updating data in a database. The actions available in the genie are shown in the image below.



The genie dynamically creates the SQL query based on the selections you make on the tabs for selecting the tables, columns, filters, and sort criteria required in your query.



Only the first 100 results for a query appear in the Preview pane. Queries can be converted between portable SQL (the default query format) and native SQL and copied any time using the Copy to Clipboard feature. Alpha Anywhere also keeps track of the queries you create in the SQL History, which can be used to retrieve past queries quickly.



Left: SQL Native Syntax dialog. **Right:** SQL History dialog.

3.11 Xbasic SQL Helper Functions

Many functions exist for performing common SQL queries using Xbasic. Internally, these functions use the SQL objects we've discussed previously to connect to and execute commands on a SQL database. The examples below show some of the more common SQL helper functions used in Xbasic scripts. You can learn more about these functions as well as other SQL helper functions in the Alpha Anywhere documentation. See SQL Helper Functions[13] for more information.

sql_lookup()

The sql_lookup()[14] function retrieves a value from a table.

```
DIM connection AS C = "::Name::AADemo-Northwind"
DIM table AS C = "customers"
DIM result_expression AS C = "concatenate(city,' - ',contactname)"
DIM filter AS C = "customerid = :whatcustomerid"
DIM args AS SQL::arguments
args.set("whatcustomerid","ALFKI")

? sql_lookup(connection,table,filter,result_expression,args)
= "Berlin - Maria Anders"
```

sql_records_get()

The sql_records_get()[15] function retrieves one or more records and returns it as a character list.

```
DIM connection AS C = "::Name::AADemo-Northwind"
DIM table AS C = "customers"
DIM result_expression AS C = "concatenate(city,' - ',contactname)"
DIM filter AS C = "city = 'London'"

? sql_records_get(connection,table,filter,result_expression)
= London - Thomas Hardy
London - Victoria Ashworth
London - Elizabeth Brown
London - Ann Devon
London - Simon Crowther
London - Hari Kumar
```

[13] https://documentation.alphasoftware.com/index?search=api%20sql%20helper%20functions
[14] https://documentation.alphasoftware.com/index?search=api%20sql%20lookup%20function
[15] https://documentation.alphasoftware.com/index?search=api%20sql%20records%20get%20function

sql_query()

The sql_query()[16] creates and executes a SQL query to select records from a SQL database.

```
DIM result AS P
DIM sql AS C = "SELECT FIRST 10 * FROM customers"
DIM cnIn AS C = "::Name::AADemo-Northwind"

result = sql_query(cnIn, sql)

? result.error
= .F.

? result.json
= [
{"CustomerID" : "ALFKI"},
{"CustomerID" : "ANATR"},
{"CustomerID" : "ANTON"},
{"CustomerID" : "AROUT"},
{"CustomerID" : "BERGS"},
{"CustomerID" : "BLAUS"},
{"CustomerID" : "BLONP"},
{"CustomerID" : "BOLID"},
{"CustomerID" : "BONAP"},
{"CustomerID" : "BOTTM"}
]
```

sql_insert()

The sql_insert()[17] function creates and executes an `INSERT` statement to add a new record to a table. The value to set in the field for a new record is defined as a JSON object of name-value pairs.

```
DIM cn AS C = "::Name::AADemo-Northwind"
DIM tablename AS C = "customers"
DIM fieldsValuePairs AS C = <<%str%
{
    "customerid":"SMPL2",
    "companyname":"ABC Co.",
    "city":"Springfield",
    "country":"USA"
}
%str%

p = sql_insert(cn,tablename,fieldsValuePairs)
? p.error
= .F.
```

[16] https://documentation.alphasoftware.com/index?search=api%20sql%20query%20function
[17] https://documentation.alphasoftware.com/index?search=api%20sql%20insert%20function

sql_update()

The sql_update()[18] function creates and executes an `UPDATE` statement to modify an existing record in a table.

```
DIM connection AS C = "::Name::AADemo-Northwind"
DIM tablename AS C = "customers"
DIM primaryKey AS C = "customerid"
DIM primaryKeyValue AS C = "SMPL2"
DIM fieldsValuePairs AS C = <<%str%
{
    "city":"Framingham"
}
%str%

p = sql_update(connection,tablename,fieldsValuePairs,primaryKey,primaryKeyValue)
? p.error
= .F.

? p.rowsAffected
= 1
```

sql_count()

The sql_count()[19] function returns a count of the number of records matching a SQL query.

```
DIM cn AS SQL::Connection
? cn.open("::Name::AADemo-Northwind")
= .T.

DIM table AS C
DIM fields AS C
DIM filter AS C
table = "customers"
' * indicates all fields in the table
fields = "*"
filter = "country='UK'"

? sql_count(cn,table,fields,filter)
```

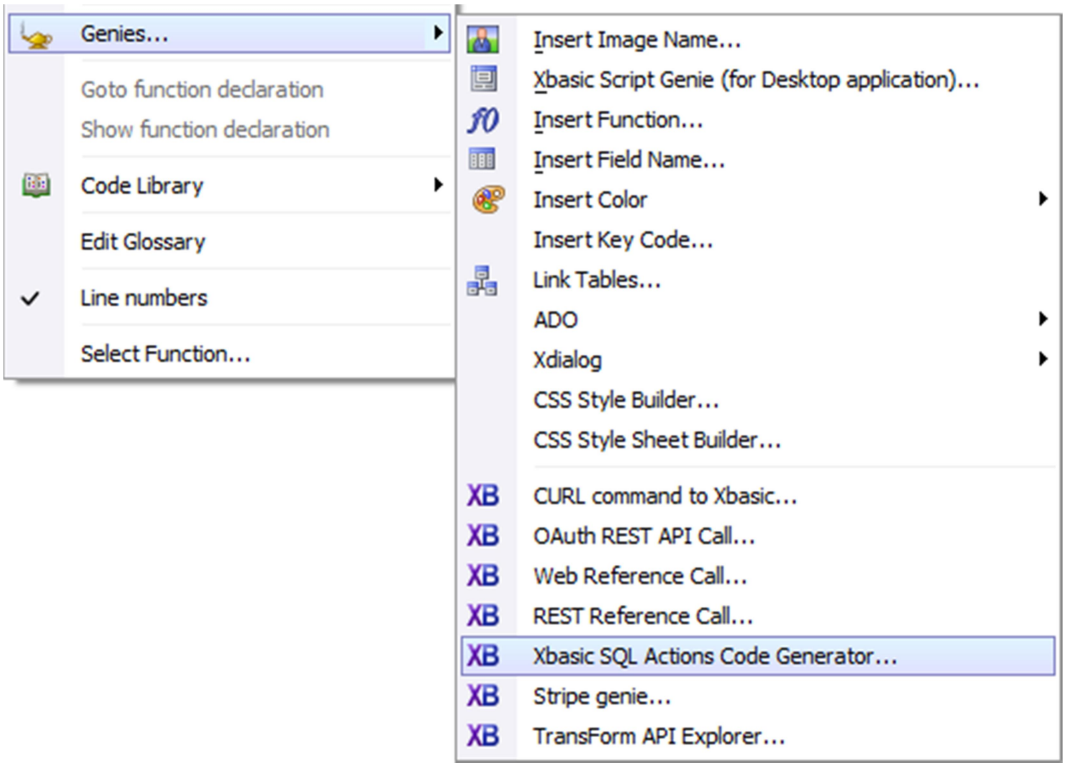
[18] https://documentation.alphasoftware.com/index?search=api%20sql%20update%20function
[19] https://documentation.alphasoftware.com/index?search=api%20sql%20count%20function

3.12 Other Helpful Tools

3.12.1 Xbasic SQL Actions Code Generator

The Xbasic SQL Actions Code Generator can be used to generate Xbasic for performing create, read, update, and delete (CRUD) operations against a data source. The Xbasic SQL Actions Code Generator is especially useful when writing server-side logic in web applications where data needs to be in a JSON format.

The genie uses AlphaDAO connection strings to connect to the data source. To access the genie, right-click anywhere in the Xbasic editor to open the context menu. Then, select "Genies..." > "Xbasic SQL Actions Code Generator..." to open the genie.



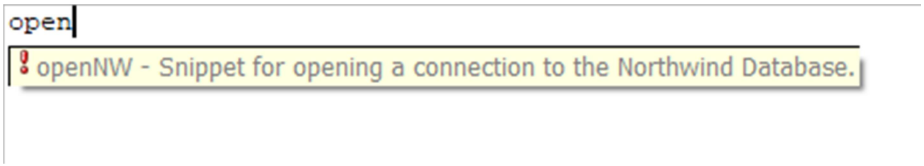
The genie generates Xbasic for the actions listed below:

- **Query a SQL database to get JSON data** – Generates the Xbasic to fetch one or more records from a table and convert the query result into a JSON object. This action, as well as the **Query a SQL database to get hierarchical JSON data**, is ideal for getting data from a database used to populate a List control, ViewBox, or other controls that require data to be in a JSON format.
- **Query a SQL database to get hierarchical JSON data** – Generates the Xbasic to fetch records from multiple tables and merge the results into a nested JSON object where child records are included as an object array property of the parent record.
- **Perform an UPDATE action** – Generates the Xbasic required to update one or more records in a table.

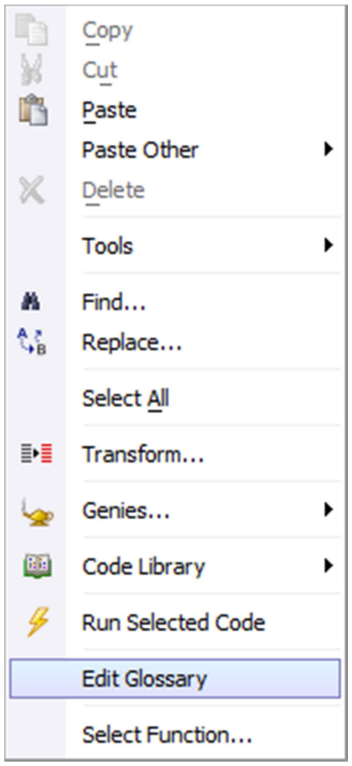
- **Perform an INSERT action** – Generates the Xbasic required to add one or more records into a table.
- **Perform a DELETE action** – Generates the Xbasic required to delete one or more records in a table.

3.12.2 Xbasic Code Glossary

The Glossary can be used to save Xbasic snippets that you frequently use. Snippets are saved with an abbreviation. When you type the abbreviation in the Xbasic editor, Alpha Anywhere automatically inserts the Xbasic snippet into your script.



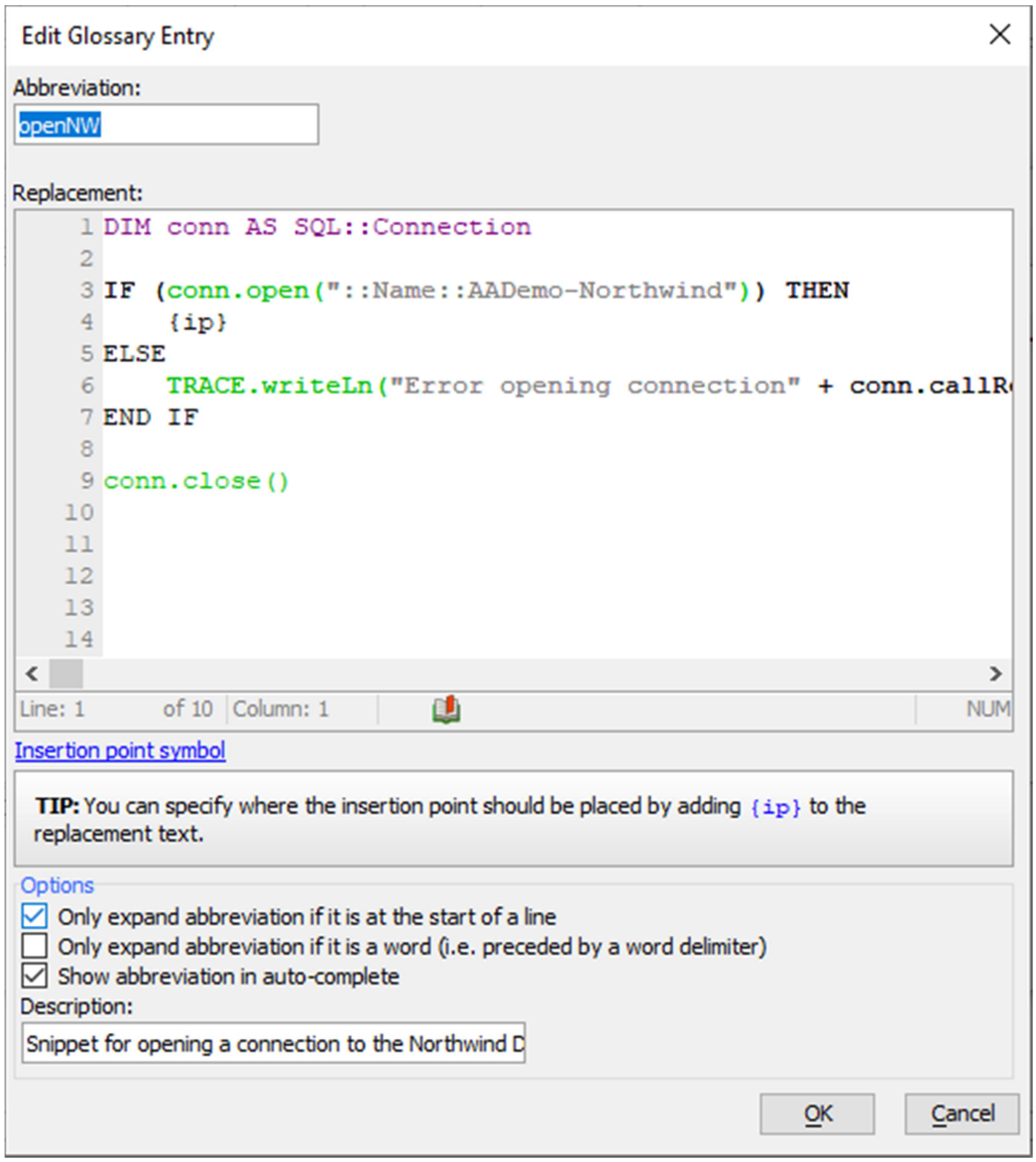
To access the Glossary, right-click in the Xbasic editor and select "Edit Glossary" from the context menu.



The Xbasic snippet can be as long as you require. The special placeholder `{ip}` defines where to place the text insertion pointer after inserting the snippet. For example:

```
DIM conn AS SQL::Connection

IF (conn.open("::Name::AADemo-Northwind")) THEN
     {ip}
ELSE
     TRACE.writeLn("Error opening connection" + conn.callResult.text,"SQL Log")
END IF

conn.close()
```

Additional options are available for configuring whether or not the snippet should appear in the auto help while you are writing Xbasic as well as restrictions on where a snippet can be inserted.



4 Calling Xbasic Scripts in Your Applications

Web and mobile applications execute Xbasic using Ajax callbacks. Ajax callbacks allow you to retrieve data from or send data to the server without needing to reload an application. Examples of operations that use Ajax callbacks include uploading a file, fetching the next set of records to display in a Grid component, creating and downloading a report, or saving data to a database.

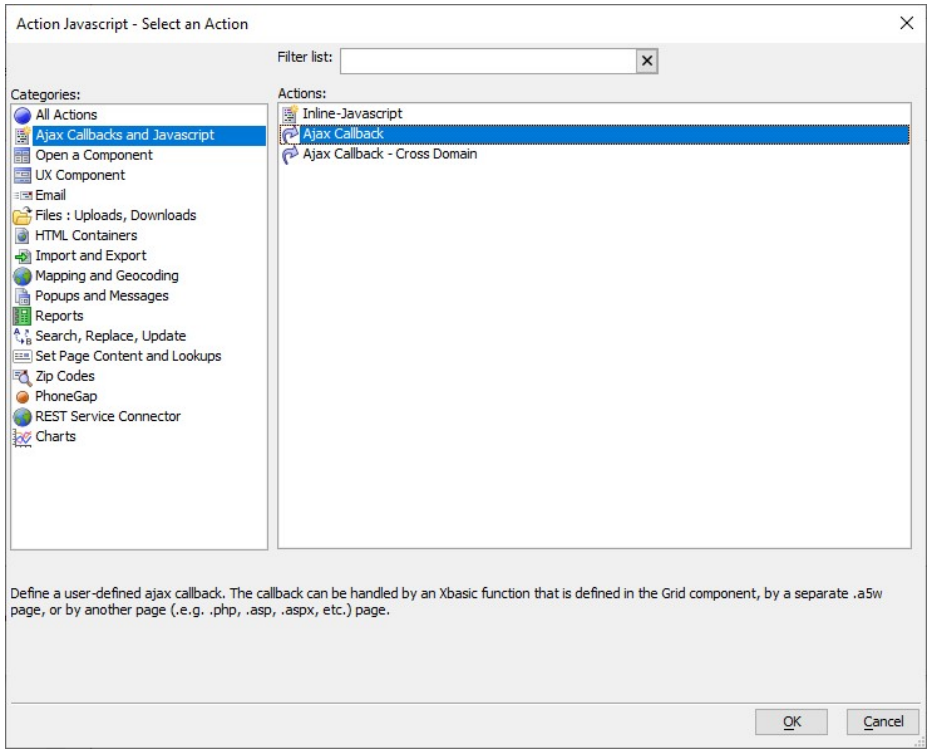
4.1 How does an Ajax Callback work?

An Ajax callback is an asynchronous exchange of messages between the client application (a web browser or mobile device) and the Alpha Anywhere Application Web Server. The user performs an action that triggers an Ajax callback. The callback is done by making a JavaScript function call that includes the Xbasic script to call and any additional information you would like to send to the server. After the callback is made, the application continues executing -- it does not wait for a response.

When the server receives the callback, it executes the requested server-side Xbasic script. After execution completes, a response containing JavaScript to execute is sent back to the calling application. When the application receives the response, it runs the JavaScript from the server.

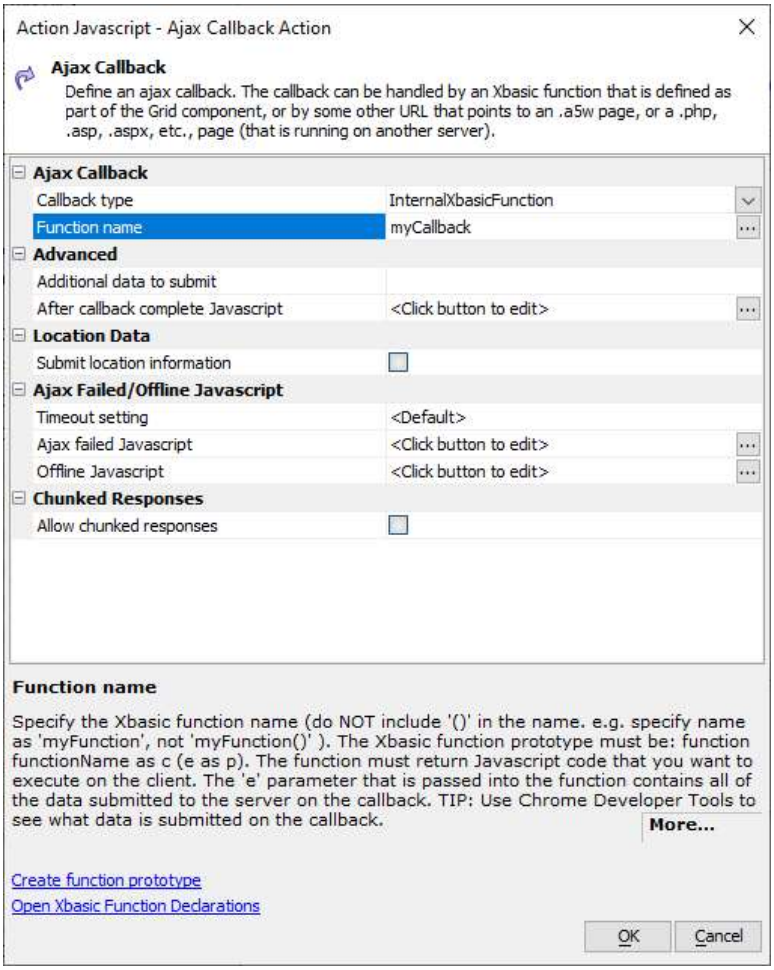
4.2 Where are Ajax Callback Functions Defined

A Callback Function is the Xbasic script executed when you make an Ajax callback. The function is commonly defined in the component's Xbasic Functions section during application design. To the call the Xbasic function, an Ajax Callback Action can be added to a control (such as a button) using the Action Javascript builder.

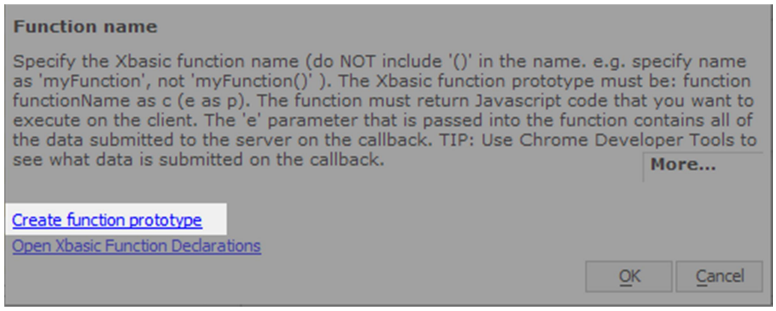


The Ajax Callback Action requires, at a minimum, the name of the Xbasic function to call. The Ajax Callback Action builder includes two helpful links for generating the Xbasic function for the callback: Create function prototype and Open Xbasic Function Declarations.

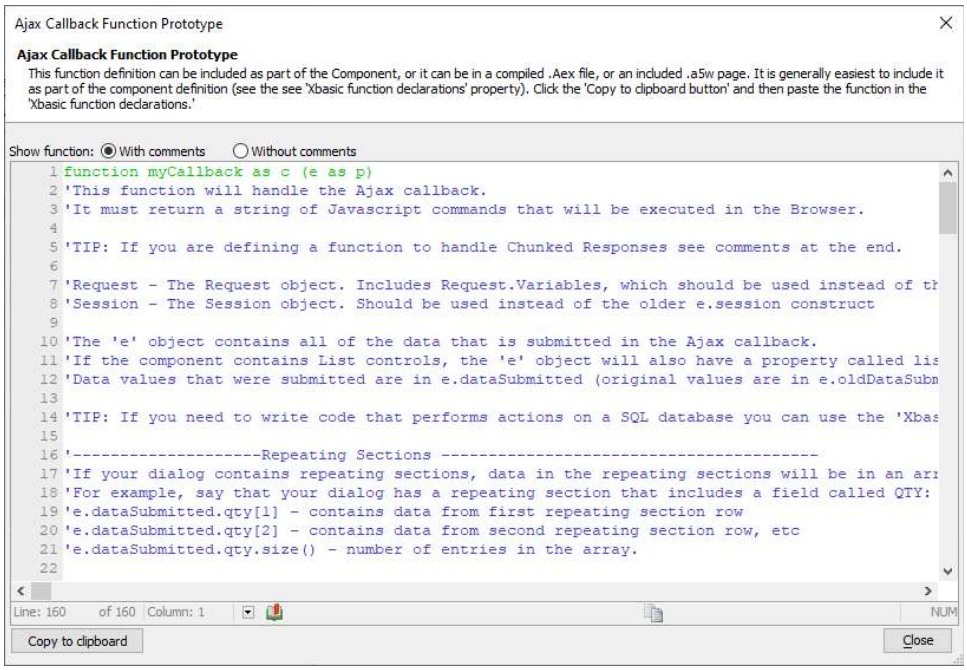
To generate the Xbasic function, you must first give the function a name -- defined in the Function name property.



Once you have named the function, you can then click on the Create function prototype to generate the Xbasic callback function.

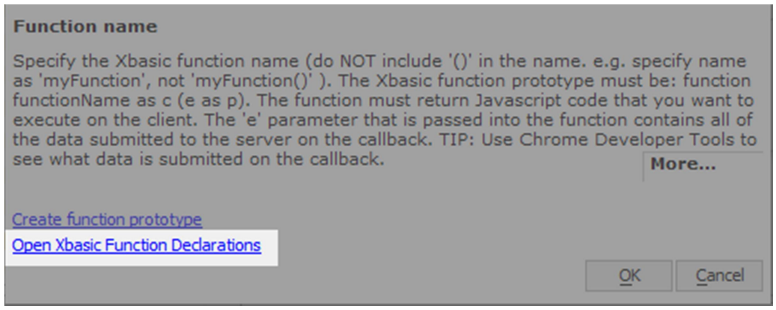


Create function prototype generates the Xbasic required for the Ajax callback, including code comments that provide information about the arguments passed to the function, available system objects (such as session), and expected return values.



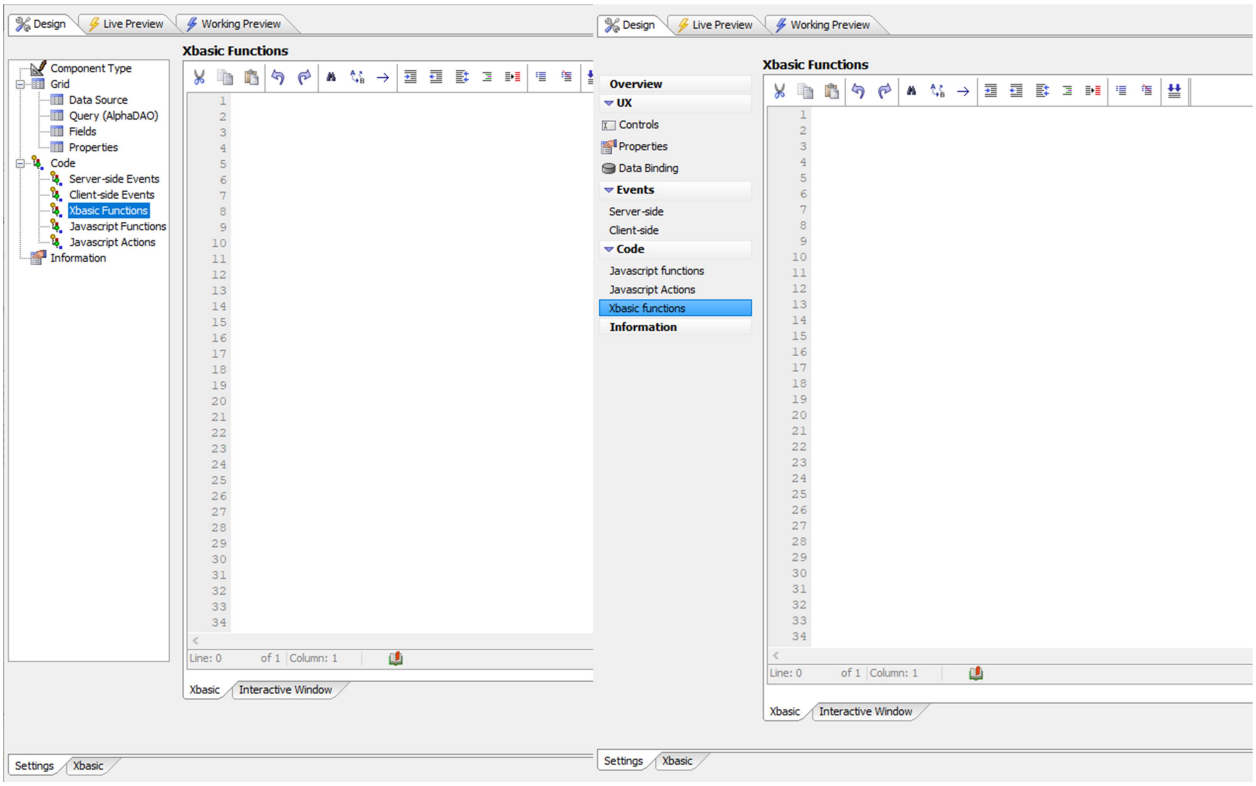
When the Function prototype appears, use the "Copy to Clipboard" button at the bottom of the Ajax Callback Function Prototype dialog to copy the Xbasic then close the dialog.

Then, click on "Open Xbasic Function Declarations" to access the Xbasic Functions section of the component.



The Xbasic Functions section is where you can define Xbasic functions for use in Ajax callbacks and server-side events. Paste the code into the Xbasic editor, and then close the editor and save your Action Javascript.

You can modify your Xbasic Function on the Xbasic Functions pane of the component. The image below shows where to find the Xbasic Functions pane in the Grid and UX Components.



Left: Grid Component Xbasic Functions pane. **Right:** UX Component Xbasic Functions pane.

4.3 Server-side Events

Xbasic is used in component server-side events. Server-side events are special Xbasic functions called when the component code executes on the server. For example, when a UX component is first loaded, the Xbasic onDialogInitialize server-side event is called.

Ajax callbacks also trigger server-side events. For example, when the `{dialog.object}.submit()` function is called, which makes an Ajax callback to submit data from the UX component to the server, it triggers the following server-side events in the order shown:

- canAjaxCallback
- onDialogExecute
- dialogValidate
- afterDialogValidate
- afterAjaxCallback

Some common use cases for server-side events include data validation, saving data to a database, customizing the component layout, calling a stored procedure, or computing initialization data, such as the choices for dropdown boxes or radio buttons.

4.3.1 Server-side Events Exercise: Populating a Dropdown Box

A dropdown control's choices can be dynamically populated using an Xbasic variable. The variable can be a session variable or a variable created in the onDialogInitialize server-side event.

Create a new blank UX Component and add a dropdown box control from the Data controls section. Open the dropdown box control's Choices and select the Variable option in the Define Choices dialog. Enter "dropdownChoices" in the Variable name box and click OK to save the settings.

Next, click on Server-side in the left-hand menu to open the Server-Side events. Select the onDialogInitialize event. Add a new line on line 2 and paste the following script into the Xbasic editor:

```
DIM places AS C = <<%txt%
Canberra,Australia
Brasilia,Brazil
Ottawa,Canada
Santiago,Chile
Copenhagen,Denmark
Tokyo,Japan
Mexico City,Mexico
Rabat,Morocco
Wellington,New Zealand
Oslo,Norway
Lima,Peru
Vanuatu,Port Vila
Cape Town,South Africa
Stockholm,Sweden
Harare,Zimbabwe
%txt%

e.rtc.dropdownChoices = places
```

Save the component and run it in Live or Working Preview. Click on the dropdown box to display the list of choices.

4.4 Persisting Data Beyond the End of a Callback

Thus far, we have only discussed creating local variables. A local variable in Xbasic only exists for the duration of the script in which the variable was created. When the script finishes execution, local variables cease to exist.

In an application, there are many cases where you may want to store data on the server beyond the end of the script's execution, such as the logged in user's name, the name of a recently uploaded file, or custom settings for the application. This type of information can be saved between callbacks using session variables.

4.4.1 What are Session Variables

Session variables are stored in the `context.session` object. The `context` object is a global Xbasic object available to any Xbasic script throughout a web application. It contains the session object where you can create variables to store data for the current user's session.

4.4.2 Creating Session Variables

Session variables are created by adding a property to the `context.session` object. For example:

```
context.session.country = "Canada"
```

All session variables are character type variables. In order to store data such as dates, numbers, arrays, or object pointers in a session variable, they must be converted to a character string.

For simple data types (such as dates, time, or logical), the easiest way to convert a value to a character is to concatenate it with an empty character value. For example:

```
DIM number AS N = 5
DIM string AS C
string = "" + number
context.session.myNumber = string
```

Complex objects such as arrays and object pointers need to be serialized to a character string before they can be stored in the session. Serializing data is done using the json_generate()[20] method to convert the array or pointer variable to a JSON string. For example:

[20] https://documentation.alphasoftware.com/index?search=json_generate%20function

```
DIM person AS P
person.name = "John Smith"
person.city = "Boston"
person.state = "MA"

' Serialize p to a character string
context.session.obj = json_generate(person)
? context.session.obj
= {
    "name": "John Smith",
    "city": "Boston",
    "state": "MA"
}
```

⚠ What about GLOBAL and SHARED variables?

If you have visited the online documentation on declaring Xbasic variables, you may have noticed that variables can be declared as `GLOBAL` or `SHARED`. These keywords will create Xbasic variables that can be accessed outside of an Xbasic function. The only way to create variables that exist beyond the end of a script in desktop applications is using the `GLOBAL` or `SHARED` keywords.

In web applications, however, `GLOBAL` and `SHARED` variables do not behave in the same way. `GLOBAL` and `SHARED` variables only exist for the duration of the Ajax callback in web applications. Once the callback completes, `GLOBAL` and `SHARED` Xbasic variables no longer exist. Because of this, you must store data that you would like to reference in other callbacks and scripts in session variables.

4.4.3 Reading Session Variables

Session variables can be read from the `context.session` object. For example:

```
DIM usercontact AS C
IF (variable_exists("context.session.userEmail")) THEN
    usercontact = context.session.userEmail
END IF
```

When you read the value from the session variable, you can use the convert_type()[21] method to cast the data to the desired type. `convert_type()` allows you to convert any data type to any other data type. If you have stored numeric values in a session variable, you would use `convert_type(context.session.myNumber,"N")` to cast the session variable to a numeric type. For example:

[21] https://documentation.alphasoftware.com/index?search=CONVERT_TYPE%20Function

```
DIM num2 AS N
' Convert session.myNumber to a number
num2 = convert_type(context.session.myNumber, "N")
```

If the session variable was created by serializing an array or object pointer using the `json_generate()` function, you can convert the json object back to a pointer variable with json_parse()[22]:

```
DIM person2 AS P
' Deserialize context.session.obj to a pointer variable
person2 = json_parse(context.session.obj)

? person2
= city = "Boston"
name = "John Smith"
state = "MA"
```

4.4.4 Session Variable Availability

Before using a session variable, you must verify that the session variable exists. Attempting to use a session variable that doesn't exist, or was deleted because the session is no longer active, will result in a runtime error. You can prevent this error by using the variable_exists()[23] function to test that the session variable is available before trying to use it.

```
DIM myNumber AS N
IF (variable_exists("context.session.myNumber") THEN
    myNumber = convert_type(context.session.myNumber,"N")
END IF
```

When a session variable is created, it is available for the duration of the session. The session will remain active as long as the following conditions are met:

- The Application Server is not stopped or restarted.
- The next interaction (page request or update) occurs within the Application Server within the session lifetime interval. The minimum value for this interval is 300 seconds (5 minutes); the interval is set in the Application Server settings.
- The session is not reset or abandoned.

4.5 The Xbasic Debugger

The `showvar()` function is useful for displaying information in the development environment. However, the `showvar()` function can only be used in the IDE. In an Ajax callback, `showvar()` is not available. This is because the Xbasic code in an Ajax callback runs on the server while the application the user interacts

[22] https://documentation.alphasoftware.com/index?search=json_parse%20function
[23] https://documentation.alphasoftware.com/index?search=api%20variable%20exists%20function

with runs in the browser (the client.) Popup windows created using Xbasic are not accessible in the browser. Instead, you need to use the Xbasic Debugger to debug the code.

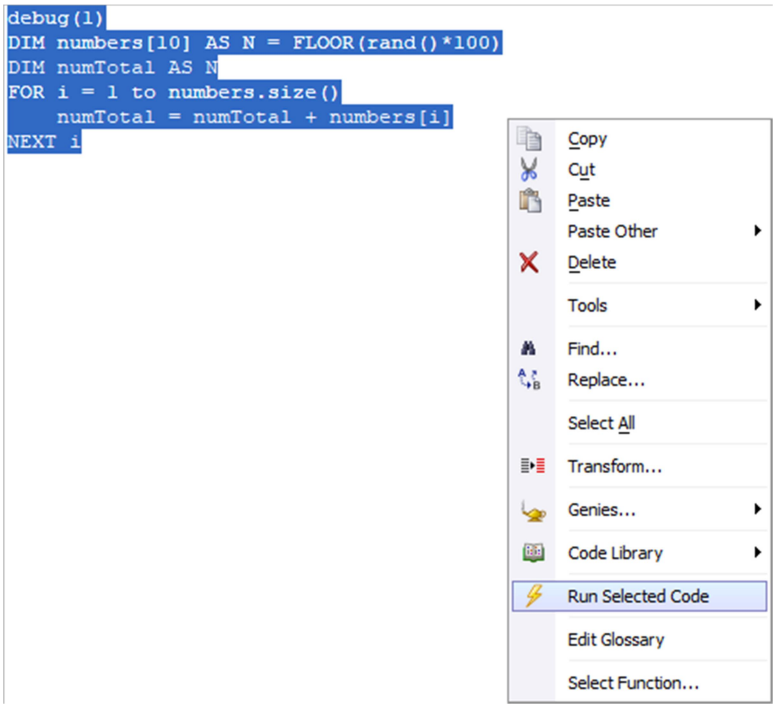
The Xbasic Debugger provides tools for inspecting your Xbasic scripts during script execution. The debugger provides standard debugging tools, such as breakpoints, watch variables, and the ability to step line-by-line through script execution.

The Xbasic Debugger is opened using the `debug()` function:

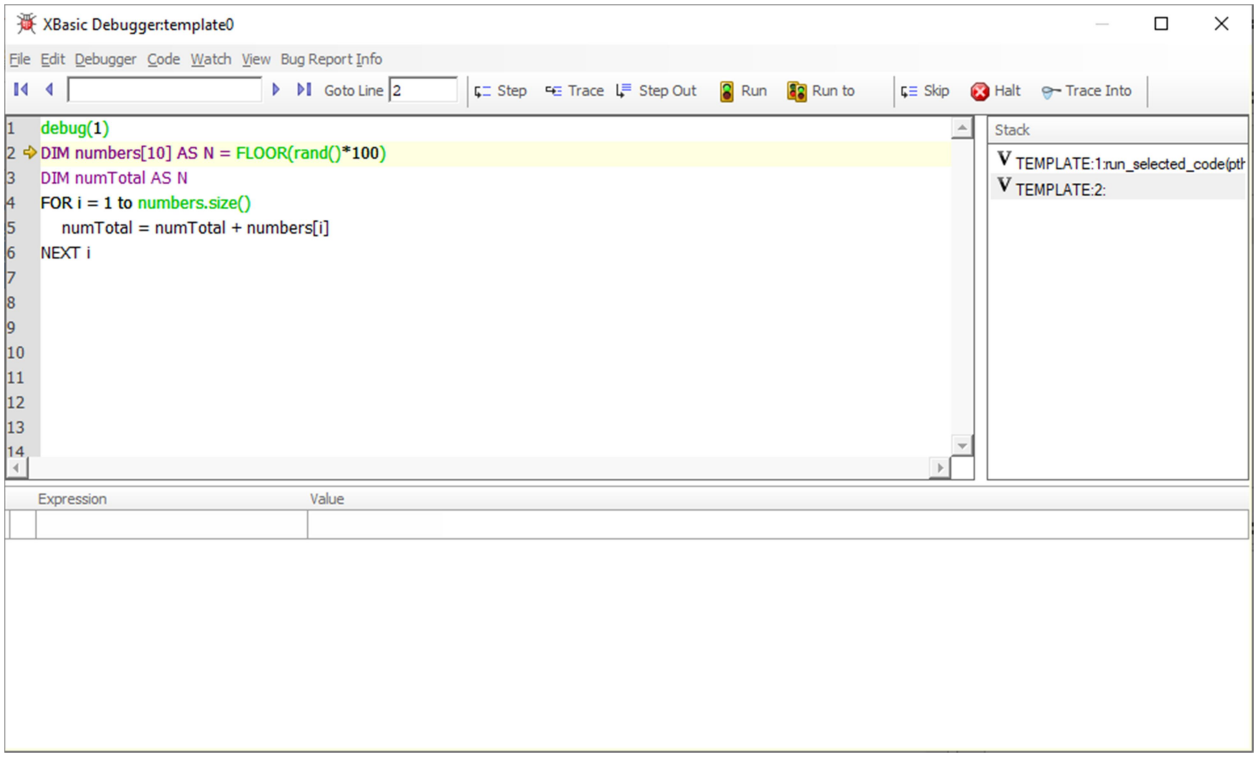
```
debug(1)
```

This statement opens the Xbasic Debugger in the Alpha Anywhere Development Environment. Let's try it out. Copy the code below into the Interactive Window. Then, select all of the code and select "Run Selected Code" from the right-click context menu.

```
debug(1)
DIM numbers[10] AS N = FLOOR(rand()*100)
DIM numTotal AS N
FOR i = 1 to numbers.size()
    numTotal = numTotal + numbers[i]
NEXT i
```



When the script executes, the Xbasic Debugger opens.

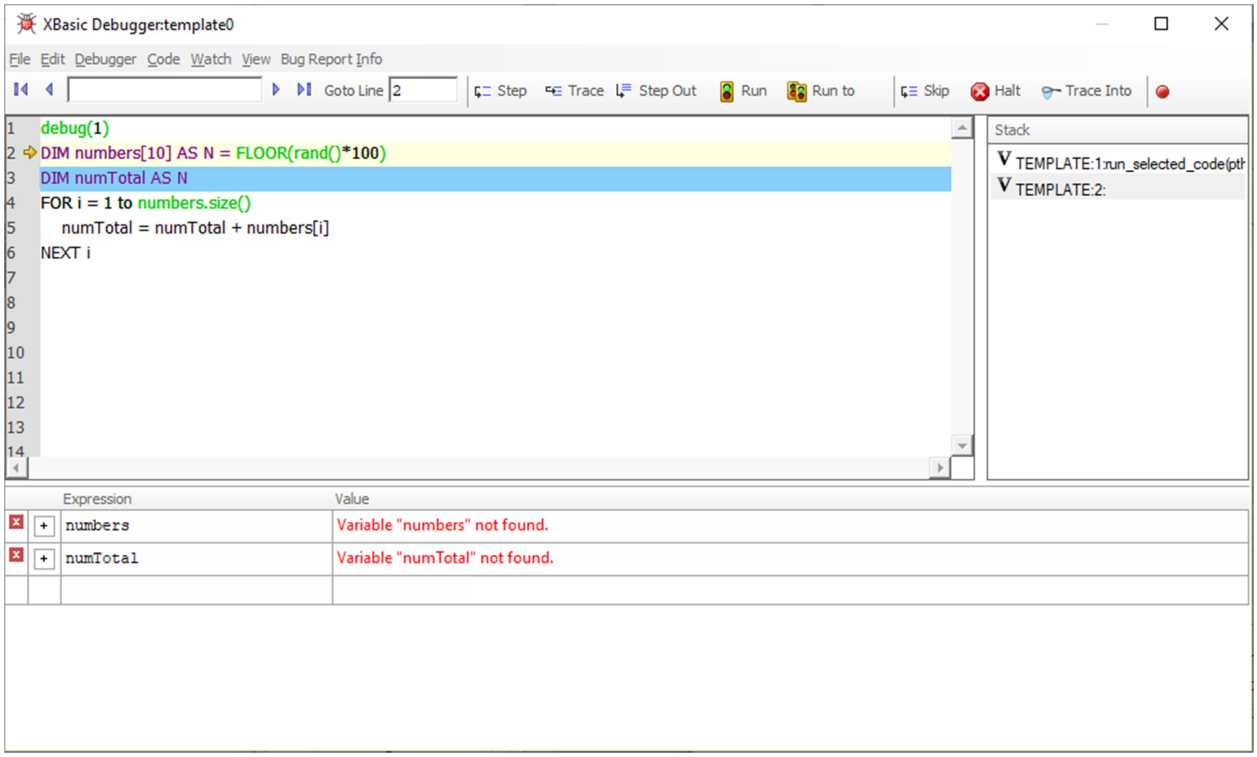


The Xbasic Debugger has three sections: the Xbasic Script, the Call Stack, and Watch Variables. The code being executed appears in the Xbasic Script portion of the debugger. The debugger highlights the next line of code to be executed with a pale yellow background.

On the right-hand side of the debugger window is the Call Stack. The Call Stack shows you the chain of functions called outside of the current script (the parent scripts.) You can mouse over the icon to the left of each entry to see the variables at that level of the stack. Clicking on an entry shows the Xbasic code that called the function.

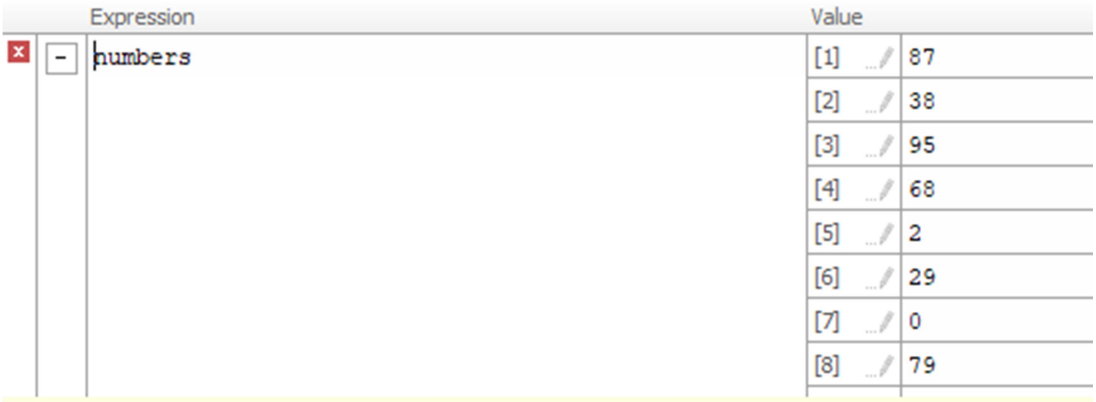
Along the bottom is the Watch window. Variables and expressions can be added to and inspected using the Watch window. As you walk through the Xbasic script in the debugger, variables in the Watch window are updated.

You can add variables to the Watch window by double-clicking the variable name or typing the variable into the Watch window. Double click the `numbers` and `numTotal` variables to add them to the Watch window.



Note that the value for both variables contains an error message: Variable not found. This error occurs because the variables do not exist. Click the Step button to advance the script and execute the Xbasic to create the `numbers` variable. Click Step again to execute the code to create the `numTotal` variable.

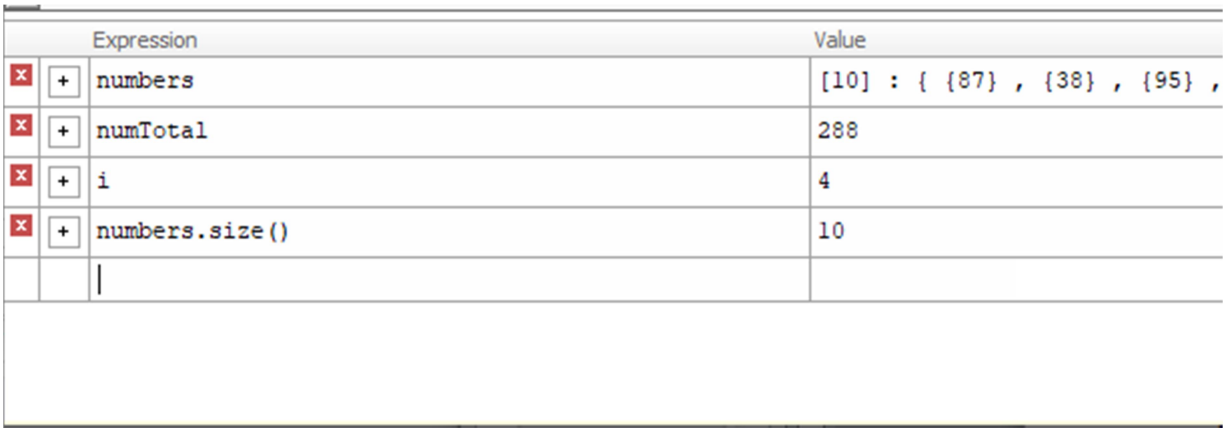
You can click the Expand/Collapse icon to expand a variable and inspect its values. For example, clicking the Expand/Collapse icon for the numbers array expands the value field, showing each entry in the array.



Clicking on one of the indexes (e.g. [3]) drills down further into the array, showing the value for that specific array element. You can use the "Set variable" icon below the Expand/Collapse icon to return to the parent object.

Add the `i` variable to the Watch window by typing it into the Expression box below the `numTotal` variable. Then, click Step to execute each line in the `FOR` loop. The Step tool loops through the `FOR` loop until the value of `i` is greater than `numbers.size()`. As you step through the loop, note that the `numTotal` and `i` variables are updated in the watch window.

You can add expressions to the watch window, as well. For example, you could add the expression `numbers.size()` to see what the size of the numbers array is:



The Xbasic Debugger contains the tools for stepping through each line of code. The standard debugger tools are Step, Trace, Step out, Run, Halt, and Set breakpoint/Clear breakpoint:

- Step – Executes the current statement.
- Trace – Executes the current statement. If the current statement includes any calls to user-defined functions, the debugger steps into the function.
- Step out – If you are inside a function call, Step out executes the rest of the function and returns to the code that called the function.
- Run – Resumes script execution.
- Halt – Terminates script execution immediately.
- Set breakpoint/Clear breakpoint – Adds or removes a breakpoint to the currently selected line. The Set breakpoint/Clear breakpoint tool appears only when a line of code is selected. You can select a line by clicking on it.

5 Learning More About Xbasic

An extensive collection of Xbasic functions, objects, and namespaces are available in the Xbasic Server-side Language Library. While we would love to cover them all here, it's beyond the scope of this document. Instead, we'll discuss the tools that are available to you to learn more about what is available in the library.

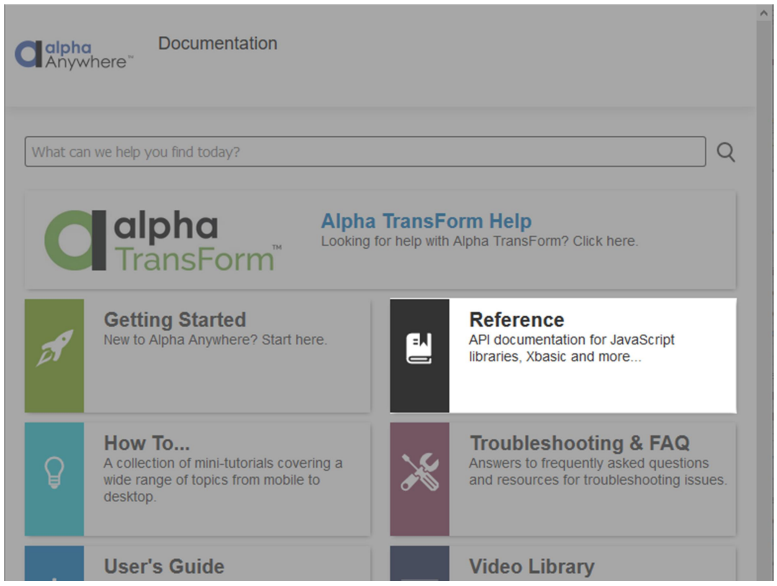
5.1 Auto-help

The Xbasic editor includes an auto-help system that offers suggestions as you write your scripts. Auto-help is useful to discover available methods for an object or namespace. It will also automatically add your user-defined functions, variables, classes, and code glossary entries, making it easier to write Xbasic scripts.

5.2 Documentation

All server-side Xbasic functions, methods, objects, namespaces, and language elements can be found in the Alpha Anywhere documentation. You can access documentation under the Help menu in Alpha Anywhere. Clicking "Open Documentation" takes you to https://documentation.alphasoftware.com.

The Xbasic language, including functions, namespaces, and objects available for use in your applications, can be found in the Reference section of the documentation.



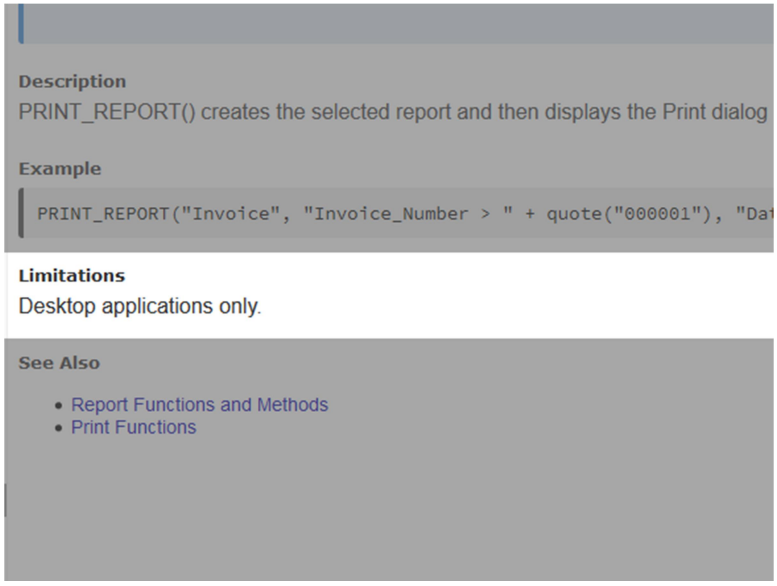
5.2.1 About the Xbasic Reference Section

The Xbasic language documentation is broken up into several sections: Server, Desktop, and Xbasic. Xbasic is the reference section for the Xbasic language. The Desktop section contains documentation for Xbasic functions and objects primarily used in Desktop applications. The Server section contains

functions, objects, and namespaces used in web/mobile applications. Many of the functions found in the Server section can also be used in Desktop applications.

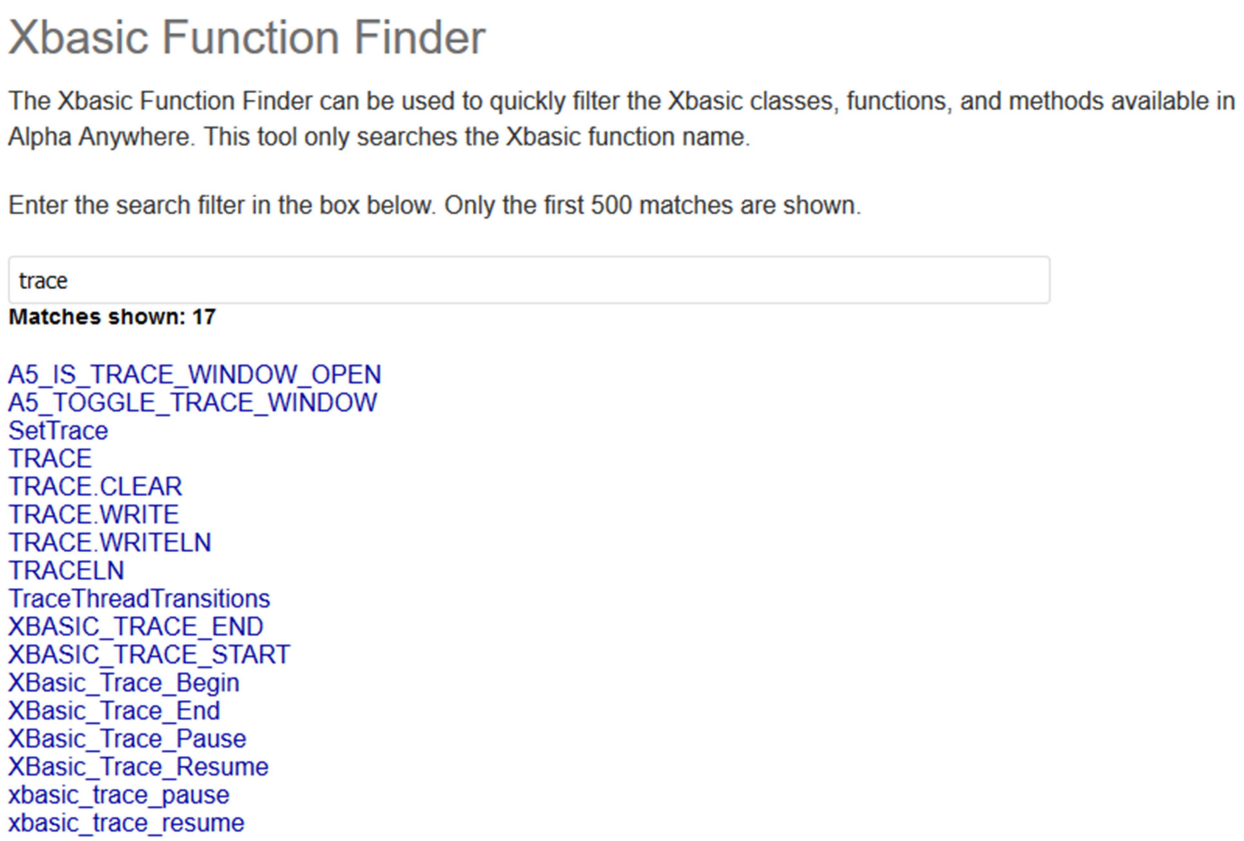
5.2.2 Limitations

Some functions in the Xbasic language library have application limitations. For example, functions marked as "Desktop applications only" cannot be used in web or mobile applications. Restricted uses for a function are listed in the limitations section at the end of the article. Not all functions include a limitations section.



5.3 Xbasic Function Finder

The Xbasic Function Finder is a tool for quickly searching functions, namespaces, and objects in the Xbasic Language library. To use the Xbasic Function Finder, navigate to https://documentation.alphasoftware.com/pages/Ref/Finder.html, type your search criteria into the search box, and hit enter. The Function Finder displays the top 500 results of your query. The finder only returns results that contain the search term in the function name.



For a more complex search, use the Documentation Search toolbar located in the upper right-hand corner of the page.

6 Appendix

6.1 Xbasic Keywords

The following words are keywords in Xbasic Language. It is highly recommended to avoid using keywords as variable names, but not required:

AS
BACKGROUND
BYREF
BYVAL
CASE
CLASS
CONSTANT
CONTINUE
CONTROL
DECLARE
DECLARESTUCT
DEFAULT
DEFINE
DELETE
DIM
EACH
ELSE
ELSEIF
END
ENUM
ERROR
EXIT
FOR
FUNCTION
GLOBAL
GOTO
IF
IMPORT
INCLUDE
INDEX
INHERITS
ITEM
LABEL
LET
LETTER
LOCAL
MACRO
MODE
NEXT
NEW
ON
OPTION
OPTIONAL
PACKAGE
PARAMS
PARENT
PARENTFORM
PRIVATE
PROTECTED
PUBLIC
READ
REDIM
RESUME
RETURN
SELECT
SERVER
SET
SHARED
STATIC
STEP
STOP
SYSTEM
THEN
THIS
TOOLBAR
TOPPARENT
TRACE
TYPE
UNDECLARE
VIRTUAL
WEBPAGE
WEND
WHILE
WITH
WRITE
YIELD