Sertainty

UXP Scripting Guide

Version: V3.6.0

Table of Contents

1 Sertainty Script

UXP Engine scripting is used to construct the authentication and governance engine. The resulting script is **KCL Code**. When a **UXP Object** is created, a **UXP Engine** script, **KCL Code,** is required and must contain a specific set of defined functions. A special set of **UXL** functions are accessible only within this mode.

The **UXL Scripting Engine** permits a user to build batch-style scripts that can perform basic utility operations. Like the **UXP Engine**, the **UXL Script Engine** has a special set of **UXL** functions that are accessible with this mode.

1.1 Sertainty Scripting Utility, UXL

The **Scripting Utility** is command line tool that supports the **UXL Script Engine**.

A sample of supported operations:

- Create a new **UXP Object**
- Add files to a **UXP Object**
- Export protected data from an existing **UXP Object**
- View the status of an existing **UXP Object**
- View the event history of an existing **UXP Object**
- Compile **UXL** source into a protected binary form

To execute the utility:

```
UXP entity-SDK-location/UXP entity [options] [command]

Options:
    -h, --help                  Displays this help.
    -b, --batch                 Execute scripting command and exit utility.
    -d, --drives                Shows current mounted drives.
    -f, --file <file>           File to include using protect.
    -g, --gui                   Use dialogs for authentication.
    -I, --id <id>               ID to use when protecting files.
    -l, --log <logfile>         Records all output into <logfile>.
    -n, --noscript              Do not invoke script engine.
    -m, --mount <UXP entity>    Mounts a UXP entity.
    -p, --protect <UXP entity>  Creates a UXP entity.
    -r, --replace               Replace output file when using protect.
    -s, --set <pref=value>      Sets a Sertainty preference.
    -v, --version               Displays version information.
    -x, --readonly              Mount UXP entity in read-only mode.

    --p1 <p1>                   Parameter passed into UXL engine.
    --p2 <p2>                   Parameter passed into UXL engine.
    --p3 <p3>                   Parameter passed into UXL engine.
    --p4 <p4>                   Parameter passed into UXL engine.
    --p5 <p5>                   Parameter passed into UXL engine.
    --p6 <p6>                   Parameter passed into UXL engine.
    --p7 <p7>                   Parameter passed into UXL engine.
    --p8 <p8>                   Parameter passed into UXL engine.
    --p9 <p8>                   Parameter passed into UXL engine.
    --p10 <p8>                  Parameter passed into UXL engine.

Arguments:
    command                     Optional UXL command to execute.
```

See **Scripting Functions** for descriptions of supported routines.

Example: Creating a new **UXP Object** using the interactive feature.

```
<Installation-location>/bin/SertaintyScript
uxl> int id;
uxl> id = sf::newUxp(“mymusic.uxp”, “myid.iic”, “IdFile”);
uxl> sf::setAttribute(id, “NAME”, “Music Backup”);
uxl> sf::setAttribute(id, “DESCRIPTION”, “Backup of my purchased songs”);
uxl> sf::setAttribute(id, “OWNER”, “Greg”);
uxl> sf::addFile(id, “song1.m4a”, “song1.m4a”, 10000, 2);
uxl> sf::addFile(id, “song2.m4a”, “song2.m4a”, 10000, 2);
uxl> sf::closeUxp(id);
uxl> exit
```

Example: Creating a new **UXP Object** using the single command line feature where the above commands are stored in a script file

```
<Installation-location>/bin /SertaintyScript -b @backup.uxl
```

In the above example, one creates a new **UXP Object** from one or more music files.

1.1.1 Initialization File

When the **Script Utility** starts, it looks for the file **scriptinit.uxl** in the **Sertainty home folder**. If found, it will attempt to execute the file. The execution is performed prior to any other script operations.

1.2 UXL Language

The **UXP Scripting Language** (**UXL**) starts as a sub-set of the C programming language and adds some proprietary constructs that enable security, portability and ease-of-use. The following sections describe the differences between C and **UXL**.

1.2.1 Comments

Like C, the /* and */ comment-block is supported. **UXL** also supports the C++ style comment of //.

1.2.2 Identifiers and Keywords

Variable and procedure names are case sensitive. All keywords and built-in procedure names are case-insensitive.

Due to optimization, all identifiers are converted to internal codes. As a result, a **UXL** error may return an error message containing an incoherent variable name if an exception occurs. To produce the same error without optimization, disable optimization by setting the **ModifierNoOptimize** at compilation or **UXP Object** creation time.

1.2.3 Data Types

The following data types are supported:

Table 1 – UXL Data Types

Data Type	Description
bytearray	Variable length array of bytes. Can handle text or binary data.
int	Integers are always 64bit and signed.
float	Floating point numbers are always double precision.
string	Character strings are always variable length and have no maximum size.
date	Standard date and time
list	A list is a variable length array containing any **UXL** type or user-defined type.

Syntax:

datatype variable-name, variable-name;

Like C, variables can be declared as arrays.

All other C data types are unsupported.

All data types will attempt to automatically convert data to the correct format.

> **Example:** If an integer variable is assigned a value from a string variable, **UXL** will attempt to convert the string to an integer. If the data cannot be converted, then an error will stop **UXL** execution.

Unlike conventional language compilers, **UXL** only supports variable scoping at two levels: **global** and **procedure**.

> **Example:** A C-language procedure allows one to create a variable within a logical block. Once the block ends, the variable will be deallocated.

For **UXL**, variables can be declared at any point within a procedure; however, they will be visible for the life of the procedure.

1.2.4 User-Defined Procedures and Rules

To declare a function in **UXL**, the following syntax must be used.

For procedures and functions:

```
[ REPLACE ] PROCEDURE [domain-name::]proc-name ( optional-arg-list )
{
  statement-list
}
```

A procedure behaves like a C-language procedure in that it can have formal arguments. Unlike C, **UXL** supports untyped arguments. An untyped argument means that a procedure can be passed any type variable without compiler restrictions. For example, a procedure expects one parameter. In typical C-language cases, the argument is a hard-wired type such as a string or a number, but in **UXL**, the data type can be evaluated at runtime. In **UXL**, the procedure can make decisions based on the data type of the argument.

As with argument passing, the **UXL** procedure can return any data type as a return argument.

For rules, the following syntax applies:

```
[ REPLACE ] RULE [domain-name::]rule-name ( )
{
  inference-statement-list
  statement-list
}
```

A rule is a special type procedure that accepts no formal arguments. It also can declare inference dependencies on other rules as well as data.

Inference is defined by the following statement syntax:

```
DEPENDS ON [RULE | PROCEDURE]  ( SUCCESS | ERROR | NOACTION | ENDOFDATA)
  {
    ON MATCH { statement-list }
    ON NOMATCH { statement-list }
    BEFORE INFERENCE { statement-list }
    AFTER INFERENCE { statement-list }
  }

DEPENDS ON DATA expression
  {
    ON MATCH { statement-list }
    ON NOMATCH { statement-list }
    BEFORE INFERENCE { statement-list }
    AFTER INFERENCE { statement-list }
  }
```

When a rule contains a dependency clause, the actual body of the rule will not execute until the dependencies are met. Conversely, if a rule or procedure is declared as a dependent, then execution may implicitly execute if the parent rule requires.

Note: Dependent rules will only execute if their current execution status is **NoAction**.

1.2.5 Macros and Conditional Compilation

The **UXL** language compiler supports several compile-time options that provide template features.

#ifdef

Syntax: `#ifdef` UXL-variable-name

UXL-code

#endif

Description: If **UXL-variable-name** exists, then compile all code that occurs until the next **#endif** occurs. The **#ifdef** and **#endif** must be the first item on the line to be valid.

#ifndef

Syntax: `#ifndef` UXL-variable-name

UXL-code

#endif

Description: If **UXL-variable-name** does not exist, then compile all code that occurs until the next **#endif** occurs. The **#ifndef** and **#endif** must be the first item on the line to be valid.

#define

Syntax: `#define` UXL-variable-name

Description: Defines the specified **UXL-variable-name** for the purpose of conditional compilation. The variable will not be compiled into the executable **UXL** code. The **#define** must be the first item on the line to be valid.

#undefine

Syntax: `#undefine` UXL-variable-name

Description: Undefines the specified **UXL-variable-name** for the purpose of the conditional compilation. The **#undefine** must be the first item on the line to be valid.

When compiling **UXL** source code, dynamic elements can be substituted for fixed placeholders. Using the **$(var-name)** token, the **UXL** compiler will substitute the contents of **var-name** for the entire placeholder. The substituted element can be anything that forms a valid **UXL** construct after substitution.

Example:

```
procedure test()
{
  String a = $(external-def-value);
}
```

To work properly, the user must pre-define the **UXL** variable **external-def-value** and populate it with a value that forms a value string assignment. In this example, one could assign the variable the value:

```
"My value"
```

After compilation, the procedure would look like:

```
procedure test()
{
  String a = "My value";
}
```

Macro variables can be defined as part of the compilation process using the C++ API routine **uxp::sys::compileUXL.**

1.2.6 Built-In Variables

The following variables are automatically defined by the system:

Table 7 – Built-In Variables

Variable Name	Data Type	Description
error	Integer	Contains the value indicating error.
errorcode	Integer	After a procedure call, this will contain the error code. A zero indicates no error has occurred.
errorstring	String	After a procedure call, this will contain the error message. An empty string indicates no error has occurred.
false	Integer	Contains the value 0.
locale	String	Contains the current locale for the current access.
missing	Integer	Contains the value indicating missing value.
missing_str	String	Contains the value indicating missing string value.
no_action	Integer	Contains the value indicating no action.
not_found	Integer	Contains the value indicating data not found.
p1	String	Contains a parameter that was passed into the engine from an application or the command line.

Variable Name	Data Type	Description
p2	String	Contains a parameter that was passed into the engine from an application or the command line.
p3	String	Contains a parameter that was passed into the engine from an application or the command line.
p4	String	Contains a parameter that was passed into the engine from an application or the command line.
p5	String	Contains a parameter that was passed into the engine from an application or the command line.
p6	String	Contains a parameter that was passed into the engine from an application or the command line.
p7	String	Contains a parameter that was passed into the engine from an application or the command line.
p8	String	Contains a parameter that was passed into the engine from an application or the command line.
p9	String	Contains a parameter that was passed into the engine from an application or the command line.
p10	String	Contains a parameter that was passed into the engine from an application or the command line.
procedure_name	String	Contains the name of the currently executing procedure.
success	Integer	Contains the value indicating success.
true	Integer	Contains the value 1.

1.2.7 Intrinsic Functions

The following procedures are native to the **UXL** language and are available for general scripting.

1.2.7.1 Intrinsic Function Summary

Table 8 – Function Summary

Function	Description
abort	Logs a fatal error and then exits the process.
abs	Determines the absolute value of an expression.
addMinutes	Adds the specified number of minutes to the date.
appendList	Appends the evaluated expression to the list.
bitTest	Determines if the bit at the specified offset is set in the number.
chr	Converts the specified expression to a string.
clear	Clears a target variable.
clearList	Removes all elements from the list.
concat	Concatenates all arguments into a single string.
copyright	Returns the product copyright declaration.

Function	Description
countList	Gets the length of the specified list.
execute	Dynamically executes a **UXL** expression.
exp	Determines the exponential value of an expression.
formatDate	Formats a date and time value using formatting rules.
getenv	Gets an environment variable.
getList	Gets an item from the specified list.
hash	Computes a hash of the specified expression.
isByteArray	Tests the data type of the expression as a bytearray type.
isDate	Tests the data type of the expression as a date type.
isFloat	Tests the data type of the expression as a float type.
isInt	Tests the data type of the expression as an integer type.
isList	Tests the data type of the expression as a list type.
isString	Tests the data type of the expression as a string type.
isStruct	Tests the data type of the expression as a structure type.
ln	Determines the natural log value of an expression.
locate	Searches for the first occurrence of string in a source string.
log	Determines the $\log^{10}$ value of an expression.
max	Extracts the maximum value from a list of values.
memcpy	Copies source data to a target variable.
memset	Fills the specified variable with an expression.
Min	Extracts the minimum value from a list of values.
osPlatform	Returns the current operating system name.
pow	Calculates a number raised to the specified power.
print	Evaluates and prints the specified expression to standard output. This is more of a utility function to support **UXL** development.
printf	Prints a formatted string to the current standard output.
printLog	Prints a formatted string to the current application log.
procedureExists	Determines if the procedure exists.
productName	Returns the current product name.
removeList	Removes the specified element number from the list.
rnd	Generates a random number.
showStats	Prints the results to the current output stream.
sizeOf	Gets the number array elements for the specified expression.
sleep	Pauses execution for the specified number of seconds.
split	Parses a source string into a list of string tokens using the specified separator.

Function	Description
sprintf	Writes a formatted string to the specified output variable.
sqrt	Determines the square root value of a numeric expression.
startStats	Starts a counter that keeps track of elapsed and cpu time.
stopStats	Stops the counters from a previous call to startStats ().
strcat	Appends the input value to an output variable.
strcpy	Copies the input value to an output variable.
strlen	Gets the length of the source string.
substr	Extracts a substring from a source string.
toUpper	Converts the specified string to uppercase.
timeDiff	Gets the number of seconds between the two date / time expressions.
timeOffset	Gets the UTC zone offset in minutes.
timeZone	Gets the current time zone.
toDate	Converts the expression to a valid date and time.
today	Returns the current date and time in UTC.
toLower	Converts the specified string to lowercase.
toString	Converts the specified expression to a string.
Trim	Trims all leading and trailing whitespace from the string.
value	Clones the evaluated expression.
variableExists	Determines if the named variable exists.

1.2.7.2 Intrinsic Function Descriptions

abort (expr)

Logs a fatal error and then exits the process.

Parameters:

expr	The message to display upon exit.

Returns:

None

varying abs (arg)

Determines the absolute value of an expression.

Parameters:

arg	Argument to process.

Returns:

Absolute value. The data type will be identical to the original argument.

date addMinutes (date , minutes)

Adds the specified number of minutes to the date.

Parameters:

date	Date / time to amend.
minutes	Number of minutes to add.

Returns:

Modified date.

appendList (list , expr)

Appends the evaluated expression to the list.

Parameters:

list	List to receive a copy of the data.
expr	Expression that produces a value to append to the list. If the expression is a variable name, the variable is cloned to produce an independent copy of the data.

Returns:

None

string bitTest (number , offset)

Determines if the bit at the specified offset is set in the number.

Parameters:

number	The number to test.
offset	The relative bit number to test in the number. Offsets begin at zero.

Returns:

Zero if the bit is not set; otherwise, the value represented by the bit offset. For example, testing bit 2 in the number 6 will yield a return value of 2.

string chr (expr)

Converts the specified expression to a string.

Parameters:

expr	ASCII value expression to evaluate and convert.

Returns:

Converted string

clear (target-name [, count-expr])

Clears a target variable.

Parameters:

target-name	The variable to clear.
count-expr	An optional expression that specifies the number of array elements to clear.

Returns:

None

clearList (list)

Removes all elements from the list.

Parameters:

list	List to clear.

Returns:

None

string concat (arg1 , … , argn)

Concatenates all arguments into a single string.

Parameters:

argn	String to append to output string

Returns:

Concatenated string

string copyright ( )

Returns the product copyright declaration.

Returns:

Copyright as a string.

int countList (list)

Gets the length of the specified list.

Parameters:

list	List to count.

Returns:

Number of elements in the list.

execute (expr)

Dynamically executes a UXL expression.

Parameters:

expr	A string containing a valid **UXL** statement.

Returns:

None

float exp (arg)

Determines the exponential value of an expression.

Parameters:

arg	Argument to process.

Returns:

Exponential value.

string formatDate (fmt , date)

Formats a date and time value using formatting rules.

Parameters:

fmt	Date format mask. The following formatting options are supported: d — The day as number without a leading zero (1 to 31) dd — The day as number with a leading zero (01 to 31) M — The month as number without a leading zero (1-12) MM — The month as number with a leading zero (01-12) yy — The year as two digit number (00-99) yyyy — The year as four digit number These expressions may be used for the time: h — The hour without a leading zero (0 to 23 or 1 to 12 if AM/PM display) hh — The hour with a leading zero (00 to 23 or 01 to 12 if AM/PM display) m — The minute without a leading zero (0 to 59) mm — The minute with a leading zero (00 to 59) s — The second without a leading zero (0 to 59) ss — The second with a leading zero (00 to 59) z — The milliseconds without leading zeroes (0 to 999) zzz — The milliseconds with leading zeroes (000 to 999) AP — Used for AM/PM display. *AP* will be replaced by either “AM” or “PM”. Ap — Used for am/pm display. *Ap* will be replaced by either “am” or “pm”.
Date	Date to format.

Returns:

The formatted date as a string.

string getenv (expr)

Gets an environment variable.

Parameters:

expr	External environment variable to fetch.

Returns:

Variable contents.

varying getList (list , element)

Gets an item from the specified list.

Parameters:

list	List containing data.
element	The zero-based element number to retrieve. The maximum element number is the list size minus one.

Returns:

The retrieved element. The data type is based on the data type of the list element.

string hash (expr)

Computes a hash of the specified expression.

Parameters:

expr	Expression to evaluate.

Returns:

Hash value as a string.

int isByteArray (expr)

Tests the data type of the expression as a bytearray type.

Parameters:

expr	Expression to test

Returns:

1 if data type is a bytearray. 0 if it is not a bytearray.

int isDate (expr)

Tests the data type of the expression as a date type.

Parameters:

expr	Expression to test

Returns:

1 if data type is a date. 0 if it is not a date.

int isFloat (expr)

Tests the data type of the expression as a float type.

Parameters:

expr	Expression to test

Returns:

1 if data type is a float. 0 if it is not a float.

int isInt (expr)

Tests the data type of the expression as an integer type.

Parameters:

expr	Expression to test

Returns:

1 if data type is an integer. 0 if it is not an integer.

int isList (expr)

Tests the data type of the expression as a list type.

Parameters:

expr	Expression to test

Returns:

1 if data type is a list. 0 if it is not a list.

int isString (expr)

Tests the data type of the expression as a string type.

Parameters:

expr	Expression to test

Returns:

1 if data type is a string. 0 if it is not a string.

int isStruct (expr)

Tests the data type of the expression as a structure type.

Parameters:

expr	Expression to test

Returns:

1 if data type is a structure. 0 if it is not a structure.

float ln (arg)

Determines the natural log value of an expression.

Parameters:

arg	Argument to process.

Returns:

Log value.

int locate (source , search)

Searches for the first occurrence of string in a source string.

Parameters:

source	Source string.
search	String to locate.

Returns:

The zero-based offset of the location. A -1 indicates not found.

float log (arg)

Determines the $\log^{10}$ value of an expression.

Parameters:

arg	Argument to process.

Returns:

Log value.

varying max (arg1 , … , argn)

Extracts the maximum value from a list of values.

Parameters:

arg	Argument to test.

Returns:

The largest value of the arguments. The data type will be identical to the original argument.

memcpy (target-name , src-name [, count-expr])

Copies source data to a target variable.

Parameters:

target-name	The variable to receive the data.
src-name	The variable containing the data to be copied.
count-expr	An optional expression that specifies the number of array elements to process.

Returns:

None

memset (target-name , fill-expr [, count-expr])

Fills the specified variable with an expression.

Parameters:

target-name	The variable to receive the fill characters.
fill-expr	An expression that will be copied to the specified variable.
count-expr	An optional expression that specifies the number of array elements in the target variable to receive the data.

Returns:

None

varying min (arg1 , … , argn)

Extracts the minimum value from a list of values.

Parameters:

arg	Argument to test.

Returns:

The smallest value of the arguments. The data type will be identical to the original argument.

string osPlatform ( )

Returns the current operating system name.

Returns:

Operating system name.

varying pow (num, exponent)

Calculates a number raised to the specified power.

Parameters:

num	Number to multiply.
exponent	Number used to raise.

Returns:

Result value.

print expr

Evaluates and prints the specified expression to standard output. This is more of a utility function to support **UXL** development.

Parameters:

expr	The command to execute.

Returns:

None

printf (format [, arg1 … , argn])

Prints a formatted string to the current standard output.

Parameters:

format	The format string containing argument placeholders. Unlike standard printf in the C-language, place holders take the form of %1, %2, etc. A maximum of five arguments is supported.
argn	An optional expression that specifies a substitution argument. Arg1 will be substituted for the placeholder %1. Arg2 will substitute into %2, etc.

Returns:

None

printLog(format [, arg1 … , argn])

Prints a formatted string to the current application log.

Parameters:

format	The format string containing argument placeholders. Unlike standard printf in the C-language, place holders take the form of %1, %2, etc. A maximum of five arguments is supported.
argn	An optional expression that specifies a substitution argument. Arg1 will be substituted for the placeholder %1. Arg2 will substitute into %2, etc.

Returns:

None

int procedureExists (name)

Determines if the procedure exists.

Parameters:

name	Procedure to find.

Returns:

1 if procedure exists. 0 if procedure does not exist.

string productName ( )

Returns the current product name.

Returns:

Product name.

removeList (list , element)

Removes the specified element number from the list.

Parameters:

list	List containing elements.
expr	The element number to remove. Lists are zero-based arrays, so the first element will be zero and the last element will be the list size minus 1.

Returns:

None

int rnd (low , high)

Generates a random number. The current time is the random number generator seed.

Parameters:

low	Low value for random number.
high	High value for random number.

Returns:

Random number

showStats ()

Prints the results to the current output stream.

Returns:

None

int sizeOf (source)

Gets the number array elements for the specified expression.

Parameters:

source	Variable to read.

Returns:

Number of elements.

sleep (expr)

Pauses execution for the specified number of seconds.

Parameters:

expr	The number of seconds to pause.

Returns:

None

list split (src , separator)

Parses a source string into a list of string tokens using the specified separator.

Parameters:

src	The source string to be parsed.
separator	The separator string use to break the source string into tokens.

Returns:

List of strings.

sprintf (outbuf , format [, arg1 … , argn])

Writes a formatted string to the specified output variable.

Parameters:

outbuf	The variable to which the formatted string will be written.
format	The format string containing argument placeholders. Unlike standard printf in the C-language, place holders take the form of %1, %2, etc. A maximum of five arguments is supported.
argn	An optional expression that specifies a substitution argument. Arg1 will be substituted for the placeholder %1. Arg2 will substitute into %2, etc.

Returns:

None

float sqrt (num)

Determines the square root value of a numeric expression.

Parameters:

num	Argument to process.

Returns:

Result value.

startStats ( )

Starts a counter that keeps track of elapsed and cpu time.

Returns:

None

stopStats ()

Stops the counters from a previous call to startStats ().

Returns:

None

strcat (output , input)

Appends the input value to an output variable.

Parameters:

output	Variable to receive data.
input	Data to append to the output variable data.

Returns:

None

strcpy (output , string)

Copies the input value to an output variable.

Parameters:

output	Variable to receive data.
input	Data to copy to the output variable data.

Returns:

None

int strlen (source)

Gets the length of the source string.

Parameters:

source	Source string.

Returns:

Length of string

string substr (source , start , length)

Extracts a substring from a source string.

Parameters:

source	Source string.
start	Offset into source string from which substring starts. Zero is the first character.
length	Number of characters to extract.

Returns:

Extracted substring.

int timeDiff (date1 , date2)

Gets the number of seconds between the two date / time expressions.

Parameters:

date1	First date / time.
date2	Second date / time.

Returns:

Number of seconds.

int timeOffset( )

Gets the UTC zone offset in minutes.

Parameters:

None

Returns:

Number of minutes.

string timeZone ( )

Gets the current time zone.

Parameters:

None

Returns:

Time zone as a string.

date toDate (expr)

Converts the expression to a valid date and time.

Parameters:

expr	Argument to process. If the expression is numeric, it attempts to convert the value from seconds to the date/time entity. If the expression is a string, then it attempts to create a date/time entity by parsing the value.

Returns:

Date value.

date today ( )

Returns the current date and time in UTC.

Returns:

The current date and time.

toLower (string)

Converts the specified string to lowercase.

Parameters:

string	String to convert.

Returns:

None

string toString (expr)

Converts the specified expression to a string.

Parameters:

expr	Expression to evaluate and convert.

Returns:

string	Converted string

toUpper (string)

Converts the specified string to uppercase.

Parameters:

string	String to convert.

Returns:

None

trim (string)

Trims all leading and trailing whitespace from the string.

Parameters:

string	String to trim.

Returns:

None

varying value (expr)

Clones the evaluated expression.

Parameters:

expr	Expression to clone.

Returns:

Temporary variable containing a copy of the data.

int variableExists (name)

Determines if the named variable exists.

Parameters:

name	Variable to find. The variable name must not be quoted.

Returns:

1 if variable exists. 0 if variable does not exist.

1.2.8 Scripting Functions

The **UXL Script Engine** is a command line interface to the **UXP Technology**. It supports the same syntax as the internal **P-code execution engine (KCL Code executor, UXP VM engine)**, but without support for any **UXP Object** authentication and access procedures and functions.

The following coding conventions are used:

- String literal values must be enclosed in double-quotes.
- One or more function arguments that are surrounded by square brackets are considered optional.

The following built-in scripting commands are supported:

@script-file

Executes the specified file as a list of one or more **UXL** script commands.

Note: Nested script files are supported.