Style Definition: Heading 4: Font color: Auto, Don't keep with next, Don't keep lines together, Don't hyphenate

S-100 – Part 13

Scripting

Page intentionally left blank

Contents

Page intentionally left blank

13-1 Scope

This Part defines a standard mechanism for including scripting support in S-100 based products. Scripting provides for processing of S-100 based datasets via script files written in the Lua programming language.

13-2 Conformance

Scripts conforming to this part shall be implemented using version 5.1 of the Lua programming language.

13-3 Normative References

The following referenced documents are required for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including amendments) applies.

Lua 5.1 Reference Manual, https://www.lua.org/manual/5.1/

ISO 19125-1:2004, *Geographic information -- Simple feature access -- Part 1: Common architecture*

13-4 Purpose

This Part is provided to permit the normative expression and processing of rules for S-100 based products. Possible usage examples include: portrayal rules, product interoperability rules, rules for detecting navigational hazards, data validation rules, etc.

The use of scripting removes ambiguity from rule expression, ensures consistency among applications, and allows for rules to be modified or extended via catalogue updates.

13-5 Scripting Catalogue

A scripting catalogue (see Figure 1) is a collection of script files written for use within a scripting domain.

For instance, portrayal is a scripting domain. The rule files contained within a Lua Portrayal Catalogue comprise a scripting catalogue.

All scripting catalogues are guaranteed to contain the standard catalogue functions defined in clause 13-8.1. Scripting catalogues may additionally contain domain specific catalogue functions. The standard catalogue functions simplify the creation, integration, and testing of scripts within a scripting domain.

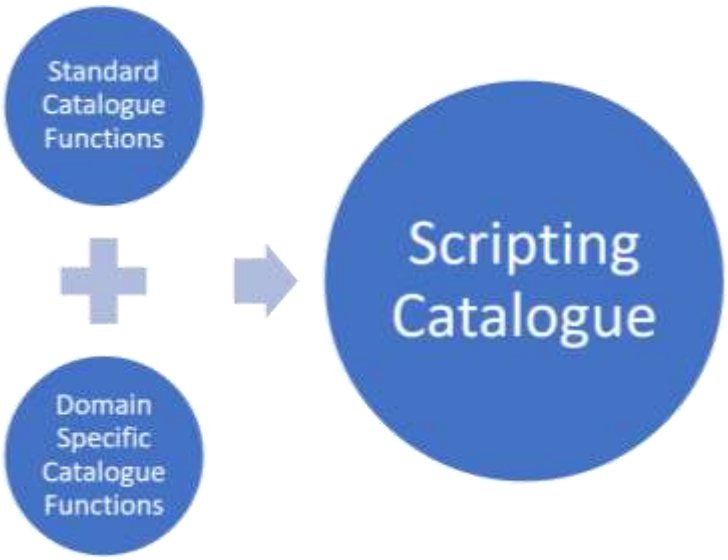

Figure 13-1 – Composition of a Scripting Catalogue

In order to apply rules within a scripting domain, scripting catalogues interact with host functions. The relationship between the scripting catalogue and the host functions is shown in Figure 13-2 below. The host functions serve to decouple the scripting catalogue from the host's implementation of S-100 concepts and functionalities.

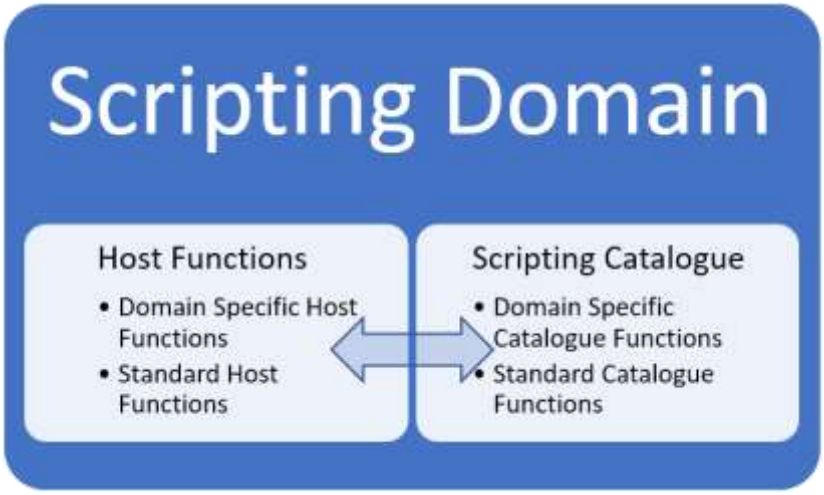

Figure 13-2 - Scripting Catalogue / Host interaction within a Scripting Domain

13-5.1 Distribution

The distribution mechanism of a scripting catalogue is defined within the scripting domain. For example, S-100 Part 9A includes a scripting catalogue within the Portrayal Catalogue; distribution of the scripting catalogue is accomplished via distribution of the Portrayal Catalogue.

Each instance of a scripting catalogue must include all standard catalogue functions.

13-5.2 Domain Specific Catalogue Functions

The standard scripting functions are always available within a scripting catalogue. Parts of S-100 which use scripting may provide additional scripting functions as needed to support domain-specific functionality. In this case, the additional functions are referred to as "domain specific functions".

Domain specific functions intended for host / scripting catalogue interaction (see Figure 2) must be specified within the relevant Part of S-100. Domain specific functions used internally within a scripting catalogue need not be specified within S-100.

For example, assume S-100 Part *N* uses scripting and requires the addition of scripting functions *X*, *Y*, and *Z*. If functions *X* and *Y* are called from the host, but function *Z* is only called from functions *X* and *Y*, S-100 Part *N* must specify required functions *X* and *Y*, and provide the documentation for each function. Since function *Z* is only used internally by the scripting catalogue, it does not need to be documented.

Domain specific functions used for interaction between a host and scripting catalogue are referred to as "domain specific host functions" or "domain specific catalogue functions", depending on whether they are implemented by the host or within the scripting catalogue.

13-6 Data Exchange

Data that is passed to the host from a scripting catalogue may be retrieved using the Lua C API functions that correspond to the data type. For the simple data types such as nil, boolean, string and number, retrieval of the data is trivial. For more complex data types, the scripting catalogue encodes the data using the Data Exchange Format (DEF) described in this section.

13-6.1 DEF Schema

The Data Exchange Format (DEF) is a string, formatted as described below. Host parsing of the DEF is simple to implement using the parsing capabilities built into all popular programming languages. Host parsing of the DEF should typically be implemented using string splitting operations such as String.split() in Java, or using simple scan parsing, such as strtok() in C or C++.

A DEF string is a series of one or more elements separated by semicolons (;). Each element is comprised of an item string, followed optionally by a colon (:) and a parameter list. A parameter list is one or more parameter strings separated by commas (,).

Note that string parameters are not surrounded by any delimiters such as quotation marks, however special characters within the string parameters will be escaped using an ampersand (&) as described in clause 13-6.1.2.

13-6.1.1 Special Characters

The following Table lists the special characters used by the DEF.

Table 13-1 – Special Characters

Special Character	Usage
Semicolon (;)	Separates the individual elements of a DEF.
Colon (:)	Separates each element into an item string and a parameter list.
Comma (,)	Separates the individual parameters of a parameter list.
Ampersand (&)	Escapes / encodes special characters contained within the DEF.

13-6.1.2 String Encoding

Special characters contained within the DEF are escaped / encoded using the character sequences listed in the following Table.

Table 13-2 – String Encoding

Special Character	Encoding
Semicolon (;)	&s
Colon (:)	&c
Comma (,)	&m
Ampersand (&)	&a

For example, a notional DEF containing four elements that might be used to represent drawing instructions:

```
PenWidth:0.64;PenColor:LANDF,0.75;DrawLine;DrawTextStrings:Hello&m world!,,Foo&cbar
```

The first element has one parameter (0.64), the second element has two parameters (LANDF and 0.75), the third element has no parameters, and the fourth element has three parameters (Hello, world!, null or empty, and Foo:bar).

13-6.1.3 Parsing

There are four steps to parsing the DEF: (1) get each element, (2) get the item and parameters for each element, (3) break the parameters into individual pieces, and then (4) decode each parameter. The notional DEF:

Item1:P1A;Item2:P2A,P2B;Item3:Hello&m world!

The host should first split the DEF into individual elements on each semicolon (;) boundary resulting in the following:

Table 13-3 – Parsing – Step 1

ELEMENT #	ELEMENT
1	*Item1:P1A*
2	*Item2:P2A,P2B*
3	*Item3:Hello&m world!*

Each of the elements should then be divided into an item and the items parameter(s) by splitting on colon (:) boundaries, resulting in:

Table 13-4 – Parsing – Step 2

ELEMENT #	ELEMENT	ITEM	PARAMETERS
1	*Item1:P1A*	*Item1*	*P1A*
2	*Item2:P2A,P2B*	*Item2*	*P2A,P2B*
3	*Item3:Hello&m world!*	*Item3*	*Hello&m world!*

The parameters should then be individually extracted by splitting the parameters on each comma (,) boundary, resulting in:

Table 13-5 – Parsing – Step 3

ELEMENT #	ELEMENT	ITEM	PARAMETER 1	PARAMETER 2	…	PARAMETER N
1	*Item1:P1A*	*Item1*	*P1A*			
2	*Item2:P2A,P2B*	*Item2*	*P2A*	*P2B*		
3	*Item3:Hello&m world!*	*Item3*	*Hello&m world!*			

Once the DEF has been divided into its constituent parts, each parameter should be converted to its original string encoding by performing the substitutions listed in Table 13-2**Error! Reference source not found.**:

Table 13-6 – Parsing – Step 4

ELEMENT #	ELEMENT	ITEM	PARAMETER 1	PARAMETER 2	…	PARAMETER N
1	*Item1:P1A*	*Item1*	*P1A*			
2	*Item2:P2A,P2B*	*Item2*	*P2A*	*P2B*		
3	*Item3:Hello&m world!*	*Item3*	*Hello, world!*			

13-6.2 Attribute Path

Scripting catalogues need to be able to determine the value of the attributes on each feature instance contained within a dataset. In order to do so, a catalogue will query the host for each attribute value as needed. When querying a host, the catalogue must identify which attribute of a given feature is being queried. If a feature instance contains only simple attributes, identifying the feature instance and attribute code is sufficient for the host to uniquely identify the requested attribute.

The host requires more information when the attribute value is contained within a complex attribute. For example, consider the following attribute value lookup:

feature.sectorCharacteristic[2].lightSector[1].valueOfNominalRange

Here the feature has a complex attribute *sectorCharacteristic*, which is an array. The second entry of *sectorCharacteristic* contains the complex attribute *lightSector*, the first entry of which contains the simple attribute *valueOfNominalRange*.

When requesting the value of *valueOfNominalRange*, scripting must provide the host with a path to the desired attribute, in addition to the *code* of the desired attribute so that the host can return the actual value. The path is required because the feature instance may have multiple attribute instances with the same *code* contained within alternate attribute paths – for example *feature.simpleAttribute*, vs. *feature.complexAttribute[n].simpleAttribute vs. feature.complexAttribute[n+1].simpleAttribute*.

When the scripting catalogue requests an attribute value from the host, an attribute path is provided to the host using a DEF string. Each section of the path is encoded as an element containing the *AttributeCode* and *Index*. *AttributeCode* contains the code of a complex attribute; *Index* stores the array index of the complex attribute.

In the example above, the path to *valueOfNominalRange* would be expressed in DEF as follows:

```
sectorCharacteristic:2;lightSector:1
```

The DEF would be used in a call to the host from a scripting catalogue as follows:

```
HostFeatureGetSimpleAttribute(featureID, sectorCharacteristic:2;lightSector:1,
valueOfNominalRange)
```

13-7 Hosting Requirements

This section defines the requirements imposed on a host in order to support scripting functionality. For example, a program written to display an S-101 ENC using the S-100 Part 9A portrayal must conform to the requirements of this section.

13-7.1 Lua Version

The host must provide a scripting engine; a Lua version 5.1 interpreter or virtual machine. The reference implementation is available from lua.org (http://www.lua.org/). Embedding the

reference implementation into the host is recommended. For maximum performance the host can embed or implement a Lua compiler such as LuaJIT (http://luajit.org/).

Further guidance on embedding is provided in *Programming in Lua – Part IV (The C API)*, details of which are available at https://www.lua.org/pil/.

13-7.2 Character Encoding

All strings exchanged between the host and the scripting catalogue must be UTF-8 encoded.

13-7.3 Error Handling

When calling Lua scripting catalogue functions from the host, a return value of **LUA_OK** from *lua_pcall* indicates success. Otherwise, the standard Lua error handling mechanism is used. An error code is returned to the host and a string detailing the error will be available on the top of the stack.

13-7.4 Array Parameters

Several of the scripting catalogue functions expect arrays to be passed as parameters. The arrays are standard Lua arrays which should be created using the Lua C API array functions as documented in *Programming in Lua – Part IV (The C API)*.

13-7.5 Host Functions

The host must provide the standard host functions detailed in clause 13-8.2.

The host must also provide domain specific host functions in order to support domain specific functionalities. Domain specific functionalities which are unused by the host do not need to be provided. Documentation for domain specific host functions is provided in the Part(s) of S-100 describing the domain specific functionality.

13-7.5.1 Compatibility

The host must guarantee backwards compatibility of the host provided functions with all previously published scripting catalogues. That is, when implementing function *X*, the host must only call scripting catalogue functions which were available in the version of S-100 when *X* was added.

Failure to conform to this requirement may result in incompatibilities when the host attempts to run older scripting catalogues.

13-7.5.1.1 Scripting Catalogue / Host Incompatibility

As new versions of S-100 are published, scripting functions may be added. Scripting functions will never be removed from S-100, although the use of a particular function may be deprecated.

Although backwards compatibility is guaranteed, newer scripting catalogues may attempt to call host functions which are unsupported by the current host. This situation is indicative of a host which has not been updated with the latest host scripting functions. To limit the occurrence of such cases, scripting catalogues should be written using the earliest subset of scripting functions possible.

Scripting incompatibilities (missing host functions) are indicated during scripting initialization. Incompatibility is indicated to the host by returning **LUA_ERRERR** from *lua_pcall*; the error string at the top of the stack will detail the cause of the incompatibility. When this occurs the host should revert to an earlier version of the scripting catalogue if available. It is also recommended to alert the user to check for an update of the host software.

13-8 Standard Script Functions

This section describes the set of standard script functions which constitute the scripting system. There are two sets of functions described: standard catalogue functions, and standard host functions. The realization of a scripting catalogue only exists within a scripting domain.

Standard catalogue functions, as described in clause 13-8.1, are provided within each scripting catalogue. Standard host functions, as described in clause 13-8.2, are to be implemented by the program which is hosting the scripting environment.

Figure 13-3 below shows the location of each type of scripting function within the scripting environment.

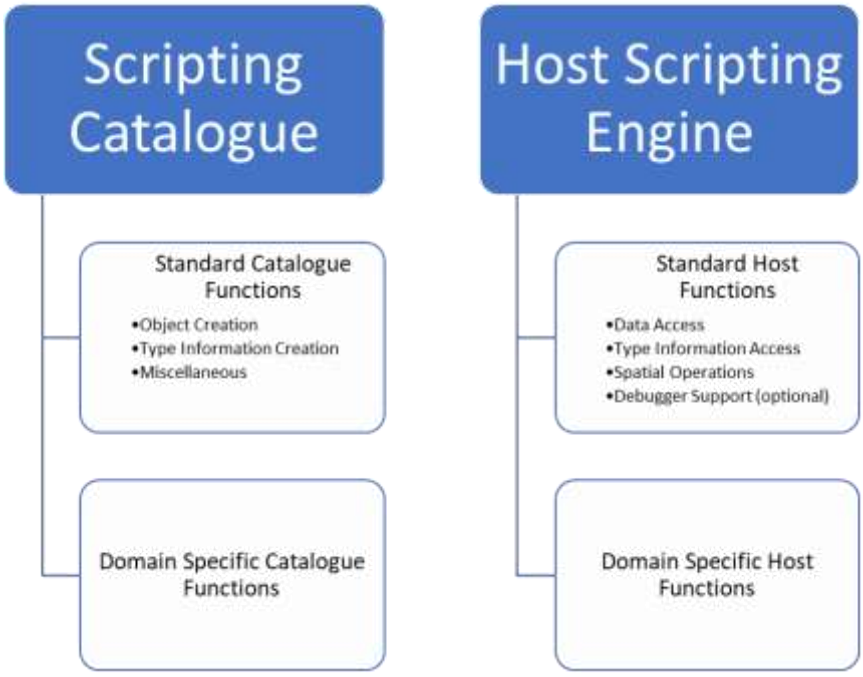

Figure 13-3 – Location of script functions within the scripting environment

Each standard script function is described below on its own sub-clause. A description of the functions purpose, along with a description of the parameters and return value are provided. For clarity, *void* is used to indicate that a function has no return value.

Function parameters which can accept multiple types will be indicated as *variant*. *variant* will also be used if the function can return more than one type. For instance, a function which accepts both integers and strings for its first parameter, and returns either an integer or string dependent on the type passed for the first parameter, would have a signature of:

variant **Function**(variant *param1*)

The function description will indicate the types which are permitted for the *variant* parameter(s).

Many of the standard script functions accept a *featureID, informationTypeID* or *spatialID* parameter. The host must ensure that these various *ID* parameters uniquely identify a single instance among all datasets across all product types to be used by the host during a scripting

session. Since each type of *ID* is a string, one way to accomplish this is by prepending the relevant information to the *ID*; for example, "S101.101US003DE01M__.000.F1" to identify the first feature in the referenced S-101 dataset.

13-8.1 Standard Catalogue Functions

This section describes the standard set of functions which are provided by all scripting catalogues.

All strings passed to these functions must be UTF-8 encoded.

When calling these functions, attribute values are always passed from the host to the scripting environment using strings. This allows values which don't have Lua equivalents to be passed unambiguously. This also allows for decimal values to be passed without the loss of precision which can occur during conversion to IEEE floating point types.

If an attribute value is present but unknown, the value returned from *GetUnknownAttributeString()* should be used.

The following Table shows the string representations of the Types defined by *S100_CD_AttributeValueType*.

Table 13-7 – String representation of types defined by *S100_CD_AttributeValueType*

S100_CD_AttributeValueType	Representation
boolean	"0" represents False "1" represents True
enumeration	S100_FC_ListedValue:code. Do not use S100_FC_ListValue:label
integer	String representation of a signed integer
real	String representation of a decimal number. Trailing zeros are permitted only if significant
text	As provided
date	Character encoding shall follow the format for date as specified by ISO 8601
time	Character encoding shall follow the format for time as specified by ISO 8601
dateTime	Character encoding shall follow the format for date and time as specified by ISO 8601
URI	Character encoding shall follow the format for URI as specified by RFC 3986
URL	Character encoding shall follow the format for URL as specified by RFC 3986
URN	Character encoding shall follow the format for URN as defined by RFC 2141
S100_CodeList	As provided
S100_TruncatedDate	As provided

13-8.1.1 Object Creation Functions

These functions relieve the host from the burden of constructing Lua tables corresponding to complex types used within the scripting catalogue. They allow the host to create objects which will be passed into the scripting catalogue. The schema and contents of the created objects are opaque to the host – they are only intended for use within the scripting catalogue.

13-8.1.1.1 AttributeConstraints CreateAttributeConstraints(integer *stringLength*, string *textPattern*, ~~integer~~string *rangeLower*, ~~integer~~string *rangeUpper*, string *rangeClosure*, integer *precision*)

Return Value

AttributeConstraints

A Lua table containing an attribute constraints object.

Parameters

stringLength: integer or nil

The maximum number of characters that may be assigned to the text attribute type. If this value is nil, the length is unconstrained.

textPattern: string or nil

A regular expression defining the structure of text values that may be assigned to the attribute. If this value is nil, the structure is unconstrained.

W3C XML Standard Part 2 Appendix F (Regular Expressions) shall be used to define the text pattern.

rangeLower: string or nil

Specifies the lower range of allowed values for the attribute. If this value is nil, there is no lower value constraint.

rangeUpper: string or nil

Specifies the upper range of allowed values for the attribute. If this value is nil, there is no upper value constraint.

rangeClosure: string or nil

Defines the closure operations for the lower and upper ranges. This is one of enumerated values as defined in Table 1-3. This must be specified if either or both the lower or upper ranges are specified.

precision: integer or nil

If specified, indicates the precision of a real number.

Remarks

Called from the host to create attribute constraints for use by the scripting catalogue.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG1]: CP1

~~13-8.1.1.1~~13-8.1.1.2 SpatialAssociation CreateSpatialAssociation(string *spatialType*, string *spatialID*, string *orientation*, variant *scaleMinimum*, variant *scaleMaximum*)

Return Value:

SpatialAssociation

A Lua table containing a spatial association object.

Parameters:

spatialType: string

The type of the spatial. One of: "Point", "MultiPoint", "Curve", "CompositeCurve", or "Surface".

spatialID: string

Used by the host to uniquely identify a spatial.

orientation: string

Orientation of the spatial. One of Forward or Reverse.

scaleMinimum: integer or nil

Minimum display scale for the spatial or nil.

scaleMaximum: integer or nil

Maximum display scale for the spatial or nil.

Remarks:

Called from the host to create a spatial association for use by the scripting catalogue.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

~~13-8.1.1.2~~13-8.1.1.3 **Point CreatePoint(string *x*, string *y*, variant *z*)**

Return Value:

Point

A Lua table containing a point object.

Parameters:

x: string

X coordinate for the point.

Y: string

Y coordinate for the point.

Z: string or nil

Z coordinate for the point. For 2D points, this value shall be *nil*.

Remarks:

x, y and *z* are expressed using the *real* string representation as described in clause 13-8.1

Called from the host to create a point spatial object for use by the scripting catalogue.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

~~13-8.1.1.3~~13-8.1.1.4 **MultiPoint CreateMultiPoint(Point[] *points*)**

Return Value:

MultiPoint

A Lua table containing a multipoint object.

Parameters:

points: Point[]

A Lua array of points. The host creates each point by calling *CreatePoint*.

Remarks:

Called from the host to create a multipoint spatial object for use by the scripting catalogue.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

~~13-8.1.1.4~~13-8.1.1.5 CurveSegment CreateCurveSegment(Point[] *controlPoints*, string *interpolation*)

Return Value:

CurveSegment

A Lua table containing a curve segment object.

Parameters:

controlPoints: Point[]

Array of points that define the control points of the curve segment. The host creates each controlPoint by calling *CreatePoint*.

Interpolation: string

The interpolation to use when connecting the control points. One of S100_CurveInterpolationL:name.

Remarks:

Called from the host to create a curve segment spatial object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

~~13-8.1.1.5~~13-8.1.1.6 Curve CreateCurve(~~Point~~ SpatialAssociation *startPoint*, ~~Point~~ SpatialAssociation *endPoint*, CurveSegment[] *segments*)

Return Value:

Curve

A Lua table containing a curve object.

Parameters:

startPoint: ~~Point~~SpatialAssociation

Start point for the curve. Host creates by calling *CreateSpatialAssociation~~CreatePoint~~*.

endpoint: SpatialAssociation~~Point~~

End point for the curve. Host creates by calling *CreateSpatialAssociation~~CreatePoint~~*.

segments: CurveSegment[]

An array of curve segments comprising the curve. Each array entry is created by calling *CreateCurveSegment*.

Remarks

Called from the host to create a curve spatial object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG2]: CP2

~~13-8.1.1.6~~13-8.1.1.7 CompositeCurve CreateCompositeCurve(SpatialAssociation[] *curveAssociations*)

Return Value:

CompositeCurve

A Lua table containing a composite curve object.

Parameters:

curveAssociations: SpatialAssociation[]

Array of spatial associations that define the elements of the composite curve. The host creates each SpatialAssociation by calling *CreateSpatialAssociation*.

Remarks:

Called from the host to create a composite curve spatial object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

~~13-8.1.1.7~~13-8.1.1.8 **Surface CreateSurface(SpatialAssociation *exteriorRing*, variant *interiorRings*)**

Return Value:

Surface

A Lua table containing a surface object.

Parameters:

exteriorRing: SpatialAssociation

The spatial association of the ring that defines the exterior ring of the surface. Host creates by calling *CreateSpatialAssociation*.

interiorRings: SpatialAssociation[] or nil

Defines the “holes” within the surface. Host creates each interior ring by calling *CreateSpatialAssociation*. If there are no holes, this parameter is nil.

Remarks:

Called from the host to create a surface spatial object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

~~13-8.1.1.8~~13-8.1.1.9 **ArcByCenterPoint CreateArcByCenterPoint(SpatialAssociation *centerPoint*, real *radius,* real *startAngle,* real *angularDistance*)**

Return Value:

ArcByCenterPoint

A Lua table containing an ArcByCenterPoint object.

Parameters:

centerPoint: SpatialAssociation

The spatial association of the point that defines the centre point of the arc. Host creates by calling *CreateSpatialAssociation*.

radius: real

Defines the geodesic distance from the centre.

startAngle: real

Starting bearing of the arc in degrees, range limited to [0.0, 360.0].

angularDistance: real

Angular distance of the arc in degrees, range limited to [-360.0, 360.0]. Positive numbers indicate a clockwise direction.

Remarks:

Called from the host to create an ArcByCenterPoint spatial object. The arc starts at the bearing given by the *startAngle* parameter and ends at the bearing calculated by adding the value of the *angularDistance* parameter to the start angle. The direction of the arc is given by the sign of the angular distance. Bearings are relative to true north except that arcs centred at either pole (where true north is undefined or ambiguous) shall use the prime meridian as the reference direction.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

~~13-8.1.1.9~~13-8.1.1.10 **CircleByCenterPoint CreateCircleByCenterPoint(SpatialAssociation *centerPoint*, real *radius*, real *startAngle,* real *angularDistance*)**

Return Value:

CircleByCenterPoint

A Lua table containing a CircleByCenterPoint object.

Parameters:

centerPoint: SpatialAssociation

The spatial association of the point that defines the centre point of the circle. Host creates by calling *CreateSpatialAssociation*.

radius: real

Defines the geodesic distance from the centre.

startAngle: real

Optional. Starting bearing of the arc in degrees, range limited to [0.0, 360.0]. Default is zero.

angularDistance: real

Optional. Angular distance of the circle in degrees, must be either -360.0 (counter-clockwise) or 360.0 (clockwise). Positive numbers indicate a clockwise direction. Default is 360 (clockwise).

Remarks:

Called from the host to create a CircleByCenterPoint object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

13-8.1.1.11 SplineCurve CreateSplineCurve(Point[] *controlPoints*, string *interpolation*, integer *degree*, Knot[] *knots*, KnotType *knotSpec*, boolean *isRational*)

Formatted: Heading 4

Return Value:

SplineCurve

A Lua table containing a spline curve.

Parameters:

controlPoints: Point[]

Array of points that define the control points of the curve segment. The host creates each controlPoint by calling *CreatePoint*. The number of control points must be three or greater.

interpolation: string

The interpolation to use when connecting the control points. One of S100_CurveInterpolation:name.

degree: integer

The degree of the polynomials used for defining the interpolation.

knots: Knot[]

Array of knots. Each knot defines a parameter in the parameter space of the spline that is used to define the spline basis function. Each knot is created by calling *CreateKnot*.

knotSpec: KnotType

Type of knot distribution in defining the spline. Defined by S100_GM_KnotType.

isRational: boolean

Indicates whether the spline uses rational functions to define the curve.

Remarks:

Called from the host to create a spline curve spatial object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG3]: CP7

13-8.1.1.12 PolynomialSpline CreatePolynomialSpline(Point[] *controlPoints*, string *interpolation*, integer *degree*, Knot[] *knots*, KnotType *knotSpec*, Vector[] *derivativeAtStart*, Vector[] *derivativeAtEnd*, integer *numDerivativeInterior*)

Formatted: Heading 4

Return Value:

PolynomialSpline

A Lua table containing a polynomial spline.

Parameters:

controlPoints: Point[]

Array of points that define the control points of the curve segment. The host creates each controlPoint by calling *CreatePoint*. The number of control points must be three or greater.

interpolation: string

The interpolation to use when connecting the control points. One of S100_CurveInterpolation:name.

degree: integer

The degree of the polynomials used for defining the interpolation.

knots: Knot[]

Array of knots. Each knot defines a parameter in the parameter space of the spline that is used to define the spline basis function. Each knot is created by calling *CreateKnot*.

knotSpec: KnotType

Type of knot distribution in defining the spline. Defined by S100_GM_KnotType.

derivativeAtStart: Vector[]

Array of Vector that defines the values used for the initial derivative used for interpolation in this curve at the start point of the spline. Up to *degree* - 2 vectors can be defined. Each vector is created by calling *CreateVector*.

derivativeAtEnd: Vector[]

Array of Vector that defines the values used for the final derivative used for interpolation in this curve at the end point of the spline. Up to *degree* - 2 vectors can be defined. Each vector is created by calling *CreateVector*.

numDerivativeInterior: KnotType

The number of continuous derivatives required at interior knots.

Remarks:

Called from the host to create a polynomial spline spatial object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG4]: CP7

13-8.1.1.13 Knot CreateKnot(string *value*[, integer *multiplicity*])

Formatted: Heading 4

Return Value:

Knot

A Lua table containing a knot object.

Parameters:

value: string

Value of the knot.

multiplicity: integer

The multiplicity of the knot. If omitted, the multiplicity is one.

Remarks:

value is expressed using the real string representation as described in clause 13-8.1

Called from the host to create a knot object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG5]: CP7

13-8.1.1.14 Vector CreateVector(Point *origin*, string[] *offset*, integer *dimension*, string *coordinateSystem*)

Formatted: Heading 4

Return Value:

Knot

A Lua table containing a knot object.

Parameters:

origin: Point

The location of the point on the GeometricReferenceSurface for which the vector is a tangent.

offset: string[]

Local tangent vector in terms of the differentials of the local coordinates. The offset values are the magnitude of the vector along each coordinate axis.

dimension: integer

The dimension of the origin.

coordinateSystem: string

The coordinate system of the origin (e.g. EPSG:4326)

Remarks:

offset values are expressed using the real string representation as described in clause 13-8.1

Called from the host to create a vector object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Formatted: Space Before: 6 pt, After: 10 pt

Commented [JM6]: CP7

Commented [DMG7]: CP7

Formatted: English (United States)

13-8.1.2 Type Information Creation Functions

These functions relieve the host from the burden of constructing Lua tables corresponding to complex types used within the scripting catalogue. They allow the host to create objects used when calling into the scripting catalogue. The schema and contents of the created objects are opaque to the host – they are only intended for use within the scripting catalogue.

The complex types correspond to the classes described in S-100 Part 5 - *Feature Catalogue*. Each type information creation function described in this section specifies the corresponding S-100 Part 5 Feature Catalogue type.

Creation functions for *FC_DefinitionReference* and its dependent types, including the *CI_Citation* class, are intentionally omitted. There are no identified use cases for *FC_DefinitionReference*, and the implementation of *CI_Citation* would be more complicated than the entirety of this section as currently defined.

13-8.1.2.1 Item CreateItem(string *code*, string *name*, string *definition*, string *remarks*, string[] *alias*)

Return Value:

Item

A Lua table containing an item corresponding to an *S100_FC_Item*.

Parameters:

code: string

Code that uniquely identifies the named type within the Feature Catalogue.

name: string

Name of the item.

definition: string

Definition of the named type in a natural language.

remarks: string

Optional. Further explanation about the item.

alias: string[]

Equivalent name(s) of this item.

Remarks:

Called from the host to create an item.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

13-8.1.2.2 NamedType CreateNamedType(Item *item,* boolean *abstract*, AttributeBinding[] *attributeBindings*)

Return Value:

NamedType

A Lua table containing a named type corresponding to an *S100_FC_NamedType*.

Parameters:

item: Item

Instance of an item created by calling *CreateItem()*. ~~Alternatively, the parameters to the *CreateItem()* function can be substituted for the single item parameter.~~

Commented [JEMJ(8]: CP8

abstract: boolean

Indicates if instances of this named type can exist in a geographic data set. Abstract types cannot be instantiated but serve as base classes for other (non-abstract) types.

attributeBindings: AttributeBinding[]

An array of zero or more bindings to attributes which describe the characteristic of this named type.

Remarks:

Called from the host to create a named type.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

13-8.1.2.3 ObjectType CreateObjectType(NamedType *namedType,* InformationBinding[] *informationBindings*)

Return Value:

ObjectType

A Lua table containing an object type corresponding to an *S100_FC_ObjectType*.

Parameters:

namedType: NamedType

Instance of a named type created by calling *CreateNamedType()*. ~~Alternatively, the parameters to the *CreateNamedType()* function can be substituted for the single *namedType* parameter.~~

Commented [JEMJ(9)]: CP8

informationBindings: InformationBinding[]

An array of zero or more bindings to information types that can be associated to this object type by means of an information association.

Remarks:

Called from the host to create an object type.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

13-8.1.2.4 InformationType CreateInformationType(ObjectType *objectType,* ~~InformationType~~ string *superType*, ~~InformationType~~string[] *subType*)

Return Value:

InformationType

A Lua table containing an information type corresponding to an S100_FC_InformationType.

Parameters:

objectType: ObjectType

Instance of a named type created by calling *CreateObjectType()*. ~~Alternatively, the parameters to the *CreateObjectType()* function can be substituted for the single *objectType* parameter.~~

Commented [JEMJ(10]: CP8

superType: ~~InformationType~~string

Optional. Indicates the code of the information type from which this type is derived.

subtype: ~~InformationType~~string[]

An array of zero or more information type codes which are derived from this type.

Remarks:

Called from the host to create an information type.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG11]: CP3

13-8.1.2.5 FeatureType CreateFeatureType(ObjectType *objectType,* string *featureUseType*, string[] *permittedPrimitives*, FeatureBinding[] *featureBindings*, ~~FeatureType~~ string *superType*, ~~FeatureType~~string[] *subType*)

Return Value:

FeatureType

A Lua table containing a feature type corresponding to an *S100_FC_FeatureType*.

Parameters:

objectType: ObjectType

Instance of a named type created by calling *CreateObjectType()*. ~~Alternatively, the parameters to the *CreateObjectType()* function can be substituted for the single *objectType* parameter.~~

Commented [JEMJ(12]: CP8

featureUseType: string

A *S100_CD_FeatureUseType:Name*.

permittedPrimitives: string[]

An array specifying zero or more allowed spatial primitive types for the feature type. Each entry is a *S100_FC_SpatialPrimitiveType:Name*.

featureBindings: FeatureBinding[]

An array of zero or more bindings to feature types that can be related to this feature type by means of a feature association.

superType: ~~FeatureType~~ string

Optional. Indicates the code of the feature type from which this type is derived. The sub-type will inherit all properties from its super-type: Name, definition and code will usually be overridden by the sub-type, although new properties may be added to the sub-type.

subType: ~~FeatureType~~string[]

An array of zero or more feature type codes which are derived from this type.

Remarks:

Called from the host to create a feature type.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG13]: CP3

13-8.1.2.6 InformationAssociation CreateInformationAssociation(NamedType *namedType,* Role[] *roles*, ~~InformationAssociation~~ string *superType*, ~~InformationAssociation~~string[] *subType*)

Return Value:

InformationAssociation

A Lua table containing an information association corresponding to an S100_FC_InformationAssociation.

Parameters:

namedType: NamedType

Instance of a named type created by calling *CreateNamedType()*. ~~Alternatively, the parameters to the *CreateNamedType()* function can be substituted for the single *namedType* parameter.~~

Commented [JEMJ(14]: CP8

roles: Role[]

An array of zero to two roles of the association.

superType: ~~InformationAssociation~~ string

Optional. Indicates the code of the information association from which this association is derived.

subType: ~~InformationAssociation~~string[]

An array of zero or more information association codes which are derived from this association.

Remarks:

Called from the host to create an information association.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG15]: CP3

13-8.1.2.7 FeatureAssociation CreateFeatureAssociation(NamedType *namedType*, Role[] *roles*, ~~FeatureAssociation~~ string *superType*, ~~FeatureAssociation~~string[] *subType*)

Return Value:

FeatureAssociation

A Lua table containing a feature association corresponding to an S100_FC_FeatureAssociation.

Parameters:

namedType: NamedType

Instance of a named type created by calling *CreateNamedType()*. ~~Alternatively, the parameters to the *CreateNamedType()* function can be substituted for the single *namedType* parameter.~~

Commented [JEMJ(16)]: CP8

roles: Role[]

An array of zero to two roles of the association.

superType: ~~FeatureAssociation~~string

Optional. Indicates the code of the feature association from which this association is derived.

subType: ~~FeatureAssociation~~string[]

An array of zero or more feature association codes which are derived from this association.

Remarks:

Called from the host to create a feature association.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG17]: CP3

13-8.1.2.8 Role CreateRole(Item *item*)

Return Value:

Role

A Lua table containing a role corresponding to a *S100_FC_Role*.

Parameters:

item: Item

Instance of an item created by calling *CreateItem()*. ~~Alternatively, the parameters to the *CreateItem()* function can be substituted for the single *item* parameter.~~

Commented [JEMJ(18)]: CP8

Remarks:

Called from the host to create a role.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

13-8.1.2.9 SimpleAttribute CreateSimpleAttribute(Item *item*, string *valueType*, string *uom*, string *quantitySpecification*, AttributeConstraints *attributeContraints*, ListedValue[] *listedValues*)

Return Value:

SimpleAttribute

A Lua table containing a simple attribute corresponding to a *S100_FC_SimpleAttribute*.

Parameters:

item: string

Instance of an item created by calling *CreateItem()*. ~~Alternatively, the parameters to the *CreateItem()* function can be substituted for the single *item* parameter.~~

Commented [JEMJ(19]: CP8

valueType: string

The value type of this feature attribute. A *S100_CD_AttributeValueType:Name*.

uom: string

Optional. Unit of measure used for values of this feature attribute. A *S100_UnitOfMeasure:Name*.

quantitySpecification: string

Optional. Specification of the quantity. A *S100_CD_QuantitySpecification:Name*.

attributeContraints: AttributeConstraints

Optional. Constraints which may apply to the attribute. Created by calling *CreateAttributeConstraints()*.

listedValues: ListedValue[]

Array of zero or more listed values for an enumerated attribute domain. Each listed value is created by calling *CreateListedValue()*. Applies only if *valueType* is *Enumeration* or *S100_Codelist* (with *codelistType* of open enumeration).

Remarks:

Called from the host to create a simple attribute type info object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

13-8.1.2.10 ComplexAttribute CreateComplexAttribute(Item *item*, AttributeBinding[] *subAttributeBindings*)

Return Value:

ComplexAttribute

A Lua table containing a complex attribute corresponding to a *S100_FC_ComplexAttribute*.

Parameters:

item: Item

Instance of an item created by calling *CreateItem()*. ~~Alternatively, the parameters to the *CreateItem()* function can be substituted for the single *item* parameter.~~

Commented [JEMJ(20]: CP8

subAttributeBindings: AttributeBinding[]

An array of one or more of attribute bindings to the sub-attributes.

Remarks:

Called from the host to create a complex attribute type info object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

13-8.1.2.11 ListedValue CreateListedValue(string *label,* string *definition*, integer *code,* string *remarks,* string[] *aliases*)

Return Value:

ListedValue

A Lua table containing a listed value corresponding to an *S100_FC_ListedValue*.

Parameters:

label: string

Descriptive label that uniquely identifies one value of the feature attribute.

definition: string

Definition of the listed value in a natural language.

code: integer

Numeric code that uniquely identifies the listed value for the corresponding feature attribute. Positive integer.

remarks: string

Optional. Further explanation about the listed value.

aliases: string[]

Optional. Array of zero or more equivalent name(s) of this listed value.

Remarks:

Called from the host to create a listed value type info object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

13-8.1.2.12 AttributeBinding CreateAttributeBinding(string *attributeCode*, integer *lowerMultiplicity,* integer *upperMultiplicity*, boolean *sequential*, integer[] *permittedValues*)

Return Value:

AttributeBinding

A Lua table containing an attribute binding corresponding to an *S100_FC_AttributeBinding*.

Parameters:

attributeCode: string

The code of the complex or simple attribute that is bound to the item or complex attribute.

lowerMultiplicity: integer

The minimum number of required occurrences of this attribute. This is zero for optional attributes.

upperMultiplicity: integer

The maximum number of allowed occurrences of this attribute. This is nil for an infinite number of allowed attributes.

sequential: boolean

Describes if the sequence of the attributes is meaningful or not. Applies only to attributes which may occur more than once.

permittedValues: integer[]

Array of zero or more permissible values of the attribute. Each entry is a *S100_FC_ListedValue:code*. Applies only to attributes of data type enumeration.

Remarks:

Called from the host to create an attribute binding object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

13-8.1.2.13 InformationBinding CreateInformationBinding(string *informationTypeCode*, integer *lowerMultiplicity,* integer *upperMultiplicity*, string *roleType*, ~~Role~~ string *role*, InformationAssociation *association*)

Return Value:

InformationBinding

A Lua table containing an information binding corresponding to a *S100_FC_InformationBinding*.

Parameters:

informationTypeCode: string

The *S100_FC_InformationType:code* of the target information type.

lowerMultiplicity: integer

The minimum number of required occurrences of this attribute. This is zero for optional attributes.

upperMultiplicity: integer

The maximum number of allowed occurrences of this attribute. This is nil for an infinite number of allowed attributes.

roleType: string

The nature of the association end. A *S100_FC_RoleType:Name*.

role: ~~Role~~string

Optional. The code of the role used for the binding. It must be part of the association used for the binding and defines the end of the association.

association: InformationAssociation

The association used for the binding; defining also the role.

Remarks:

Called from the host to create an information binding object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG21]: CP4

13-8.1.2.14 FeatureBinding CreateFeatureBinding(string *featureTypeCode*, integer *lowerMultiplicity*, integer *upperMultiplicity*, string *roleType*, ~~Role~~ string *role*, FeatureAssociation *association*)

Return Value:

FeatureBinding

A Lua table containing a feature binding corresponding to a *S100_FC_FeatureBinding*.

Parameters:

featureTypeCode: string

The code of the target feature type.

lowerMultiplicity: integer

The minimum number of required occurrences of this attribute. This is zero for optional attributes.

upperMultiplicity: integer

The maximum number of allowed occurrences of this attribute. This is nil for an infinite number of allowed attributes.

roleType: string

The nature of the association end. A *S100_FC_RoleType:Name*.

role: ~~Role~~string

The code of the role used for the binding. It must be part of the association used for the binding and defines the end of the association.

association: FeatureAssociation

The association used for the binding.

Remarks:

Called from the host to create a feature binding object.

It is not intended that the host manipulate the returned object; the object is intended to be passed from the host back to the scripting catalogue.

Commented [DMG22]: CP4

13-8.1.3 Miscellaneous Functions

The functions described on the following pages do not fall under one of the previously described functionalities.

13-8.1.3.1 string GetUnknownAttributeString()

Return Value:

string

A string that represents an attribute value that is present but unknown.

Remarks:

Intended to permit differentiation of unknown string values from empty string values. This function returns a constant value.

13-8.1.3.2 string EncodeDEFString(string *input*)

Return Value:

string

An encoding of *input* as described in clause 13-6.1.2.

Parameters

input: string

The unencoded string.

Remarks:

Encodes the input string as described in section 13-6.1.2.

13-8.1.3.3 string DecodeDEFString(string *encodedString*)

Return Value:

string

A decoded version of *encodedString*.

Parameters

encodedString: string

The encoded string.

Remarks:

Decodes an input string which was previously encoded as described in section 13-6.1.2.

13-8.1.3.4 void TypeSystemChecks(boolean *enabled*)

Return Value:

None

Parameters

enabled: boolean

Enables or disables type checking.If *true* then parameters to Lua functions are checked to ensure the parameters are of the expected type.

Remarks:

Indicates the data type of each parameter should be verified on each function call. ~~Type system checks are off~~Disabled by default. Type system checks ~~can~~may be enabled during catalogue development as a debugging aid.~~to aid in catching data type errors.~~

Commented [JEMJ(23]: CP9

13-8.2 Standard Host Functions

The host must provide a set of "callback" functions that provide the scripting environment with: Access to the host's realization of the S-100 General Feature Model; access to type information for any entity defined by the model; and access to spatial operations which can be used to perform relational tests and operations on spatial elements defined by the model. The host may optionally provide a callback function used to interact with a debugger.

Offloading these tasks to the host, rather than providing rigid data structures which are passed between the host and scripting, allows the host to interact with scripting using the hosts optimal representation of the General Feature Model. Host translation of its internal data model to a particular input schema is not necessary when using scripting.

Any of the standard host functions may be called from the scripting catalogue during the execution of a script.

13-8.2.1 Data Access Functions

The host must implement the functions described on the following pages to allow the scripting environment to access data the host has loaded from a dataset. These functions provide the scripting environment with access to features, spatials, attribute values, and information associations.

13-8.2.1.1 string[] HostGetFeatureIDs()

Return Value:

string[]

A Lua array containing all of the feature IDs in the dataset.

Remarks:

Instructs the host to return all feature IDs relevant to the current scripting catalogue operation. This would typically be all of the features in an *S100_Dataset* or *S100_DataCoverage*.

As discussed in clause 13-8, the host is responsible for ensuring each feature ID uniquely identifies a single feature instance among all product types and datasets to be used during the current scripting session.

13-8.2.1.2 string HostFeatureGetCode(string *featureID*)

Formatted: Font: Not Italic

Return Value:

string

The code defined by the Feature Catalogue for the feature type of the feature instance.

Parameters:

featureID: string

Used by the host to uniquely identify a feature instance.

Remarks:

Instructs the host to return the feature type code for the feature instance identified by *featureID*.

13-8.2.1.3 string[] HostGetInformationTypeIDs()

Return Value:

string[]

A Lua array containing all of the information type IDs in the dataset.

Remarks:

Allows scripts to query the host for a list of information types contained within a given dataset.

Instructs the host to return an array containing all information IDs in the given dataset.

13-8.2.1.4 string HostInformationTypeGetCode(string *informationTypeID*)

Formatted: Font: Not Italic

Return Value:

string

The code defined by the Feature Catalogue for the information type of the information type instance.

Parameters:

informationTypeID: string

Used by the host to uniquely identify an information type instance.

Remarks:

Instructs the host to return the information type code for the information type instance identified by *informationTypeID*.

13-8.2.1.5 string[] HostFeatureGetSimpleAttribute(string *featureID*, path *path*, string *attributeCode*)

Return Value:

string[]

The textual representation of each attribute value, as described in clause 13-8.1. An array is returned even if the attribute has a single value.

Parameters:

featureID: string

Used by the host to uniquely identify a feature instance.

path: path

An attribute path as described in clause 13-6.2

attributeCode: string

One of the attribute codes defined in the Feature Catalogue for the feature type identified by *featureID*.

Remarks:

Instructs the host to perform a simple attribute lookup on the attribute *attributeCode* at the path *path* for the feature instance identified by *featureID*. An empty array is returned if the requested attribute is not present.

13-8.2.1.6 integer HostFeatureGetComplexAttributeCount(string *featureID*, path *path*, string *attributeCode*)

Return Value:

integer

The number of matching complex attributes that exist at the path for the feature instance.

Parameters:

featureID: string

Used by the host to uniquely identify a feature instance.

path: path

An attribute path as described in clause 13-6.2.

attributeCode: string

One of the attribute codes defined in the Feature Catalogue for the feature type identified by *featureID*.

Remarks:

Instructs the host to return the number of attributes matching *attributeCode* at the given attribute path for the given feature instance. The given path will always be valid for the feature instance. The returned integer can be zero.

13-8.2.1.7 SpatialAssociation[] HostFeatureGetSpatialAssociations(string *featureID*)

Return Value:

SpatialAssociation[]

A Lua array containing all of the spatial associations for the feature instance represented by *featureID.*

Parameters:

featureID: string

Used by the host to uniquely identify a feature instance.

Remarks:

Instructs the host to return an array containing the spatial associations for the given feature instance. For each spatial association the feature contains, the host calls the standard catalogue function *CreateSpatialAssociation* to create the *SpatialAssociation* object.

The host should return an empty array if the feature has no spatial associations.

13-8.2.1.8 string[] HostFeatureGetAssociatedFeatureIDs(string *featureID*, string *associationCode*, variant *roleCode*)

Formatted: Font: Not Italic

Return Value:

string[]

A Lua array containing the associated features IDs.

Parameters:

featureID: string

Used by the host to uniquely identify a feature instance.

associationCode: string

Code for requested association as defined by the Feature Catalogue.

roleCode: string or nil

Code for requested role as defined by the Feature Catalogue. Can be nil if *associationCode* by itself is enough to specify the association or if all roles defined by *associationCode* are desired.

Remarks:

When called, the host returns an array containing the feature IDs associated with the given feature instance that match *associationCode* and *roleCode*. If no matches are found the host returns an empty array.

The *roleCode* may be nil, in which case only the *associationCode* should be used for lookup.

13-8.2.1.9 string[] HostFeatureGetAssociatedInformationIDs(string *featureID*, string *associationCode*, variant *roleCode*)

Formatted: Font: Not Italic

Return Value:

string[]

A Lua array containing the associated information IDs.

Parameters:

featureID: string

Used by the host to uniquely identify a feature instance.

associationCode: string

Code for requested association as defined by the Feature Catalogue.

roleCode: string or nil

Code for requested role as defined by the Feature Catalogue. Can be nil if *associationCode* by itself is enough to specify the association or if all roles defined by *associationCode* are desired.

Remarks:

When called, the host returns an array containing the information IDs associated with the given feature instance that match *associationCode* and *roleCode*. If no matches are found the host returns an empty array.

The roleCode may be nil, in which case only the *associationCode* is used for lookup.

13-8.2.1.10 string[] HostGetSpatialIDs()

Return Value:

string[]

A Lua array containing all of the spatial IDs in the dataset.

Remarks:

Instructs the host to return all spatial IDs relevant to the current scripting catalogue operation. This would typically be all of the spatial objects in an *S100_Dataset* or *S100_DataCoverage*.

As discussed in clause 13-8, the host is responsible for ensuring each spatial ID uniquely identifies a single spatial instance among all product types and datasets to be used during the current scripting session.

13-8.2.1.11 Spatial HostGetSpatial(string *spatialID*)

Return Value:

Spatial

A spatial object created via a standard catalogue function as listed in the remarks.

Parameters:

spatialID: string

Used by the host to uniquely identify a spatial.

Remarks:

Queries the host for a given spatial.

The host returns a spatial object created by one of the standard catalogue functions defined in clause 13-8.1.1.

13-8.2.1.12 variant HostSpatialGetAssociatedInformationIDs(string *spatialID*, string *associationCode*, variant *roleCode*)

Formatted: Font: Not Italic

Return Value:

nil

The information association is not valid for this spatial.

String[]

A Lua array containing the associated information IDs.

Parameters:

spatialID: string

Used by the host to uniquely identify a spatial.

associationCode: string

Code for requested association as defined by the feature catalogue.

roleCode: string or nil

Code for requested role as defined by the feature catalogue. Can be nil if *associationCode* by itself is enough to specify the association or if all roles defined by *associationCode* are desired.

Remarks:

When called, the host returns an array containing the information IDs for the given spatial instance that match *associationCode* and *roleCode*. If the information association is not valid for this feature according to the feature catalogue, the host returns nil. If no matches are found the host returns an empty array.

The *roleCode* may be nil, in which case only the *associationCode* is used for lookup.

13-8.2.1.13 string[] HostSpatialGetAssociatedFeatureIDs(string *spatialID*)

Return Value:

string[]

A Lua array containing the requested associated feature IDs for the spatial identified by *spatialID.*

Nil

No features are associated to the spatial identified by *spatialID.*

Parameters:

spatialID: string

Used by the host to uniquely identify a spatial.

Remarks:

When called, the host returns an array of all feature instances that reference the given spatial. A feature instance is considered to be associated to a spatial either directly through the spatial associations on the feature, or indirectly in the case of curves referenced by composite curves.

13-8.2.1.14 string[] HostInformationTypeGetSimpleAttribute(string *informationTypeID*, path *path*, string *attributeCode*)

Formatted: Font: Not Italic

Return Value:

string[] or *nil*

The textual representation of each attribute value, as described in clause 13-8.1. An array is returned even if the attribute has a single value. The host should return nil if the requested attribute is not present.

Parameters:

informationTypeID: string

Used by the host to uniquely identify an information instance.

path: path

An attribute path as defined in clause 13-6.2.

attributeCode: string

One of the attribute codes defined in the Feature Catalogue for the information type identified by *informationTypeID*.

Remarks:

Instructs the host to perform a simple attribute lookup on the attribute *attributeCode* at the indicated *path* for the information instance identified by *informationTypeID*. Nil is returned if the requested attribute is not present.

13-8.2.1.15 integer HostInformationTypeGetComplexAttributeCount(string *informationTypeID*, path *path*, string *attributeCode*)

Return Value:

integer

The number of matching complex attributes that exist at the path for the information instance.

Parameters:

informationTypeID: string

Used by the host to uniquely identify an information instance.

path: path

An attribute path as described in clause 13-6.2.

attributeCode: string

One of the attribute codes defined in the Feature Catalogue for the information type identified by *informationTypeID*.

Remarks:

Instructs the host to return the number of attributes matching *attributeCode* at the given attribute path for the given information instance. The given path will always be valid for the information instance. The returned integer can be zero.

13-8.2.2 Type Information Access Functions

These functions allow the scripting environment to query the type information for any entity from any dataset. The type information provided by the host must match the information from the relevant feature catalogue.

13-8.2.2.1 string[] HostGetFeatureTypeCodes()

Return Value:

string[]

Array containing all feature type codes as defined in the Feature Catalogue.

Remarks:

13-8.2.2.2 string[] HostGetInformationTypeCodes()

Return Value:

string[]

Array containing all information type codes as defined in the Feature Catalogue.

Remarks:

13-8.2.2.3 string[] HostGetSimpleAttributeTypeCodes()

Return Value:

string[]

Array containing all simple attribute type codes as defined in the Feature Catalogue.

Remarks:

13-8.2.2.4 string[] HostGetComplexAttributeTypeCodes()

Return Value:

string[]

Array containing all complex attribute type codes as defined in the Feature Catalogue.

Remarks:

13-8.2.2.5 string[] HostGetRoleTypeCodes()

Return Value:

string[]

Array containing all role type codes as defined in the Feature Catalogue.

Remarks:

13-8.2.2.6 string[] HostGetInformationAssociationTypeCodes()

Return Value:

string[]

Array containing all information association type codes as defined in the Feature Catalogue.

Remarks:

13-8.2.2.7 string[] HostGetFeatureAssociationTypeCodes()

Return Value:

string[]

Array containing all feature association type codes as defined in the Feature Catalogue.

Remarks:

13-8.2.2.8 FeatureType HostGetFeatureTypeInfo(string *featureCode*)

Return Value:

FeatureType

Lua data structure created by the *CreateFeatureType*() function.

Parameters:

featureCode: string

Feature code matching an entry in the Feature Catalogue.

Remarks:

13-8.2.2.9 InformationType HostGetInformationTypeInfo(string *informationCode*)

Return Value:

InformationType

Lua data structure created by the *CreateInformationType*() function.

Parameters:

informationCode: string

Information code matching an entry in the Feature Catalogue.

Remarks:

13-8.2.2.10 SimpleAttribute HostGetSimpleAttributeTypeInfo(string *attributeCode*)

Formatted: Font: Not Italic

Return Value:

SimpleAttribute

Lua data structure created by the *CreateSimpleAttribute*() function.

Parameters:

attributeCode: string

Simple attribute code matching an entry in the Feature Catalogue.

Remarks:

13-8.2.2.11 ComplexAttribute HostGetComplexAttributeTypeInfo(string *attributeCode*)

Formatted: Font: Not Italic

Return Value:

ComplexAttribute

Lua data structure created by the *CreateComplexAttribute*() function.

Parameters:

attributeCode: string

Complex attribute code matching an entry in the Feature Catalogue.

Remarks:

13-8.2.2.12 Role HostGetRoleTypeInfo(string roleCode)

Formatted: Font: Not Italic

Return Value:

Role

Lua data structure created by the *CreateRole* function

Parameters:

roleCode: string

Role code matching an entry in the feature catalogue.

Remarks:

Commented [DMG24]: CP5

13-8.2.2.13 InformationAssociation HostGetInformationAssociationTypeInfo(string informationAssociationCode)

Formatted: Font: Not Italic

Return Value:

InformationAssociation

Lua data structure created by the *CreateInformationAssociation* function

Parameters:

informationAssociationCode: string

Information association code matching an entry in the feature catalogue.

Remarks:

Commented [DMG25]: CP5

13-8.2.2.14 FeatureAssociation HostGetFeatureAssociationTypeInfo(string featureAssociationCode)

Formatted: Font: Not Italic

Return Value:

FeatureAssociation

Lua data structure created by the *CreateFeatureAssociation* function

Parameters:

featureAssociationCode: string

Feature association code matching an entry in the feature catalogue.

Remarks:

Commented [DMG26]: CP5

13-8.2.3 Spatial Operations Functions

These functions allow the scripting environment to perform relational tests and operations on spatial elements.

The host must implement the functions described on the following pages to provide the scripting environment with the ability to relate spatial entities to one another.

13-8.2.3.1 boolean HostSpatialRelate(string *spatialID1*, string *spatialID2*, string *intersectionPatternMatrix*)

Formatted: Font: Not Bold

Return Value:

boolean

Returns *true* if the geometries represented by the two spatials are related as specified in the DE-9IM matrix.

Parameters:

spatialID1: string

Used by the host to uniquely identify a spatial instance.

spatialID2: string

Used by the host to uniquely identify a spatial instance.

intersectionPatternMatrix: string

DE-9IM intersection matrix expressed as nine characters in row major order. For example, when testing for overlap between two areas: “T*T***T**”

Remarks:

Spatially relates the geometries represented by *spatialID1* and *spatialID2* using the DE-9IM intersection specified via the *intersectionPatternMatrix* string.

For details on DE-9IM string representation refer to ISO 19125-1:2004, *Geographic information -- Simple feature access -- Part 1: Common architecture, section 6.1.14.2 The Dimensionally Extended Nine-Intersection Model (DE-9IM)*.

13-8.2.4 Debugger Support Functions

These functions allow the scripting environment to interact with a debugger which may be running on the host. A debugger may be desired as an aide in developing the required standard host functions.

Host implementation of the debugger support functions is optional. Scripts will execute normally regardless of whether the host implements these functions.

13-8.2.4.1 void HostDebuggerEntry(string *debugAction*, ~~string~~ variant ~~*message*~~*parameters*)

Return Value:

None

Parameters:

debugAction: string

Indicates the requested debugger action:

break – Pause execution of the script.

t~~T~~**race** – Display a string in the debugging console, as provided in the first parameter~~.~~

S~~s~~tart_~~profiler~~ performance – Begin line by line profiling of the script code.

S~~s~~top_~~profiler~~ performance – Stop line by line profiling of the script code. The name of the performance counter is in the first parameter.

first_chance_error – Notifies the host of an impending error function call in the script. The parameter is the message passed to the error function. The second parameter is the depth passed to the error function.

~~*Message*~~*parameters*: ~~string~~variant

Zero or more parameters for use by the debug action.~~Message to display on the debugging console. This is optional for all debug actions except trace, where it is mandatory.~~

Remarks:

Host implementation of this function is optional.

Commented [DMG27]: CP6