S-100 – Part 50

Scripting

Page intentionally left blank

Contents

Page intentionally left blank

50-1 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.

50-2 Conformance

Scripts conforming to this part shall be implemented using version 5.3 of the Lua programming language.

50-3 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.

IETF RFC 7159, *The JavaScript Object Notation (JSON) Data Interchange Format.*

Lua 5.3 Reference Manual, https://www.lua.org/manual/5.3/

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

50-4 Abbreviations and Definitions

API – Application Programming Interface

Domain Specific Functions – All scripting functions which are defined outside of this part. The union of Domain Specific Host Functions and Domain Specific Catalogue Functions.

Domain Specific Catalogue Functions – Scripting functions provided within a scripting catalogue which are not part of the standard catalogue functions.

Domain Specific Host Functions – Scripting functions provided by a host to support domain-specific functionalities.

ECDIS – Electronic Chart Display and Information System

Host – The environment hosting the Lua interpreter. This is usually an application which utilizes S-100 products, such as an ECDIS.

Host Functions – The scripting functions provided by a host. The union of the Standard Host Functions and the Domain Specific Host Functions.

JSON – JavaScript Object Notation.

Scripting Catalogue – Generic term describing a collection of one or more files containing scripting functions.

Scripting Domain – The application of scripting to an S-100 domain, such as portrayal.

Scripting Engine – A Lua interpreter or virtual machine.

Scripting Function – A function written in Lua.

Standard Catalogue Functions - Scripting functions which are guaranteed to be provided within all scripting catalogues.

Standard Host Functions – Scripting functions which must be provided by the host.

Standard Scripting Functions – All scripting functions defined within this part. The union of Standard Host Functions and Standard Catalogue Functions.

50-5 Purpose

This part is provided to permit the unequivocal 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.

50-6 Scripting Catalogue

A scripting catalogue 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 50-9.1. 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.

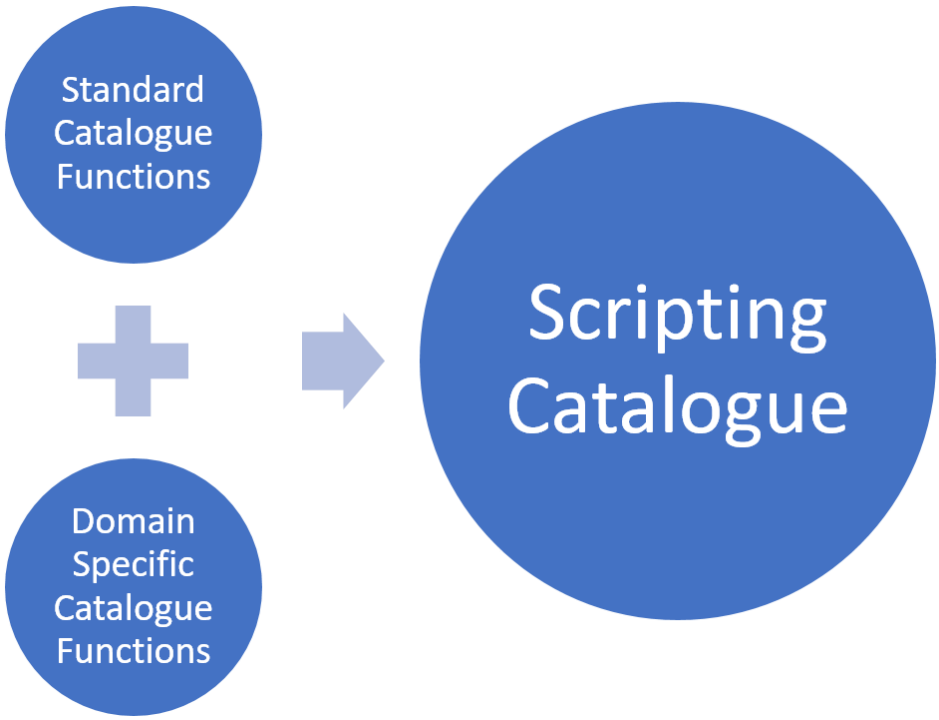

Figure 1 – Composition of a Scripting Catalogue

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 below. The host functions serve to decouple the scripting catalogue from the hosts implementation of S-100 concepts and functionalities.

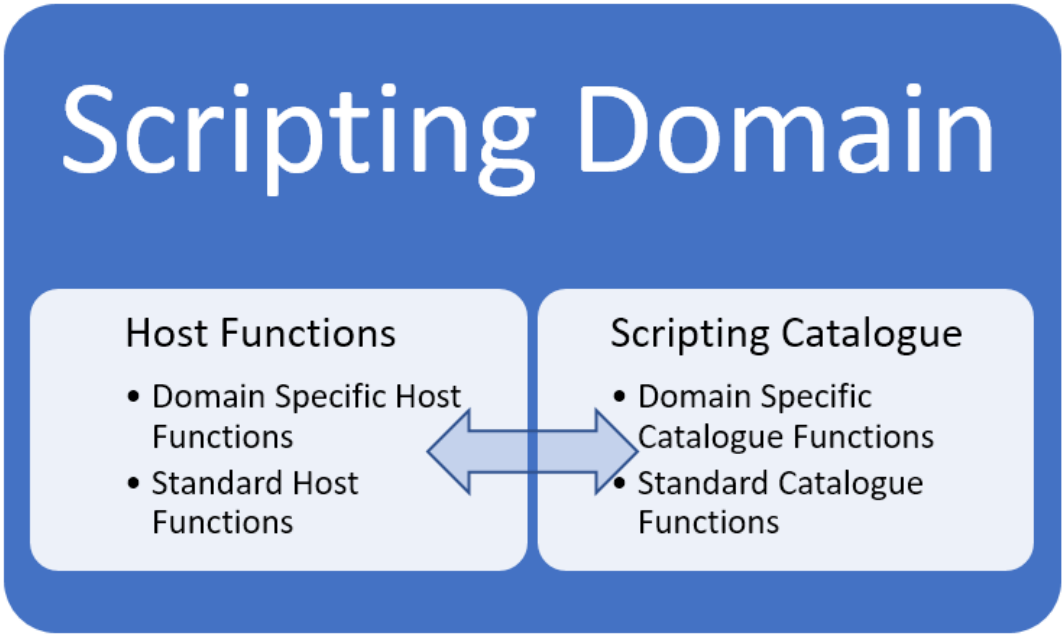

Figure 2 - Scripting Catalogue / Host interaction within a Scripting Domain

50-6.1 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.

50-6.2 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 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*. S-100 Part *N* must specify functions *X*, *Y*, and *Z* are required, and provide the documentation for each function.

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 where they are implemented.

50-7 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 boolean, string and number, retrieval of the data is trivial. However, the scripting catalogue encodes complex data types into Lua tables which the host may find more difficult to parse.

The host may find it easier to parse the complex data types using JavaScript Object Notation (JSON). To support this, the scripting catalogue provides the *ConvertToJSON* function to convert Lua table data into a JSON string.

Details of the JSON string format are provided in the documentation of the *ConvertToJSON* function. Note that passing data into the scripting catalogue using JSON is not supported.

50-7.1 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. The path is required because the feature instance may have multiple attribute instances with the same *code* contained within alternate attribute paths – e.g. *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 as an array of tables. Each table has two elements: *AttributeCode* and *Index*. *AttributeCode* contains the code of a complex attribute; *Index* stores the array index of the complex attribute.

Rather than parsing attribute paths from a Lua table, the host may use *ConvertToJSON* to express the attribute path using JSON. In the example above, the path to *valueOfNominalRange* would be expressed in JSON as follows:

```
[
      {
            "AttributeCode" : "sectorCharacteristic",
            "Index" : 2
      },
      {
            "AttributeCode" : "lightSector",
            "Index" : 1
      }
]
```

50-8 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 chart using the S-100 Part 9A portrayal must conform to the requirements of this section.

50-8.1 Lua Version

The host must provide a scripting engine; a Lua version 5.3 interpreter or virtual machine. The reference implementation is available from lua.org. Embedding the reference implementation into the host is recommended.

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/.

50-8.2 Character Encoding

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

50-8.3 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.

50-8.4 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)*.

50-8.5 Host Functions

The host must provide the standard host functions detailed in 50-9.1.

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.

50-8.5.1 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.

50-8.5.1.1 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 will 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.

50-9 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 host functions and standard catalogue functions.

Standard host functions, as described in 50-9.1, are to be implemented by the program which is hosting the scripting environment. Standard catalogue functions, as described in 50-9.1, are provided within a scripting catalogue.

The figure below shows the location of each type of scripting function within the scripting environment.

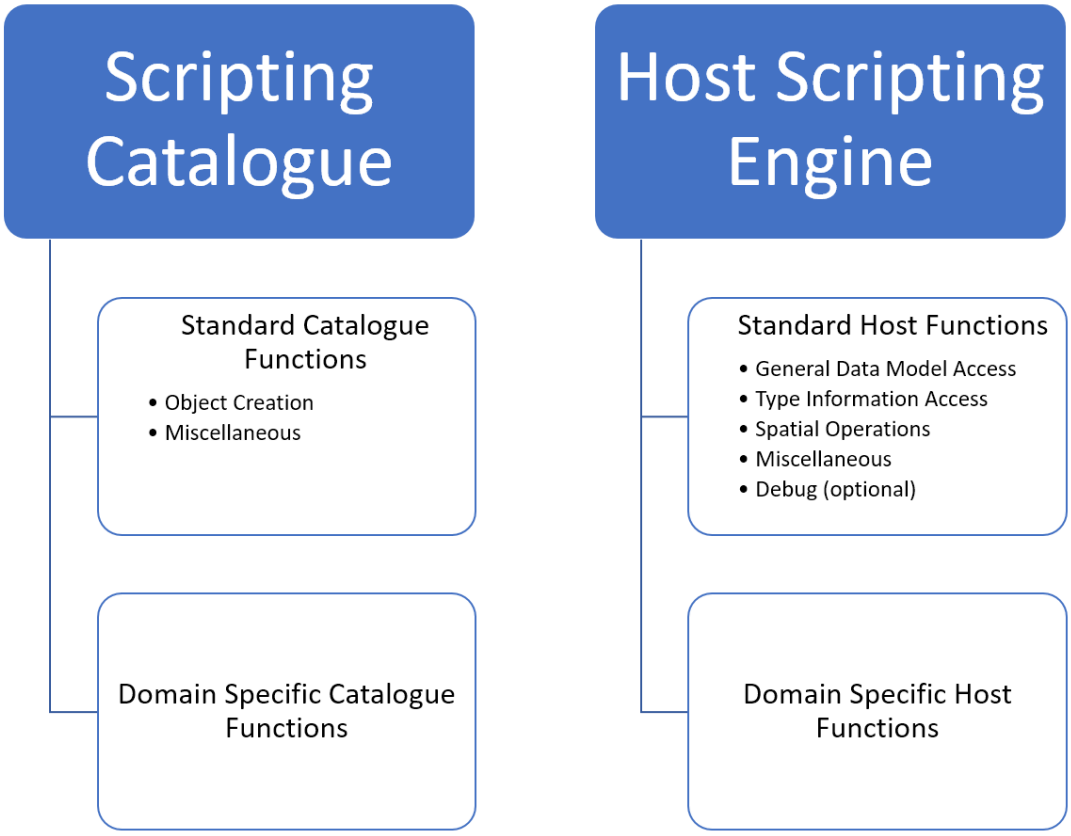

Figure 3 – Location of script functions within the scripting environment

Each standard script function is described below on its own page. 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 *datasetID*, *featureID*, or other type of *ID* 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: e.g. "S101.US3DE01M__.000.F1" to identify the first feature in the referenced S-101 dataset.

50-9.1 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.

The following table shows the string representations of the value 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.

50-9.1.1 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 used when calling into the scripting catalogue. The contents of the created objects are opaque to the host – they are only intended for use within the scripting catalogue.

> ***NOTE: The spatial object creation functions provided here are sufficient for processing of S-101 datasets. The next revision of this part will include complete coverage of the spatial types defined in S-100 Part 7.***

50-9.1.1.1 SpatialAssociation **CreateSpatialAssociation**(string *spatialType*, string *spatialID*, string *orientation*, integer *scaleMinimum*, integer *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 spatial association directly; the spatial association is intended to be passed from the host back to the scripting catalogue.

50-9.1.1.2 Point **CreatePoint**(string *x*, string *y*, string *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

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

Remarks

The *x, y* and *z* are expressed using the *real* string representation as described in section 50-9.1

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 point directly; the point is intended to be passed from the host back to the scripting catalogue.

50-9.1.1.3 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 multipoint directly; the multipoint is intended to be passed from the host back to the scripting catalogue.

50-9.1.1.4 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: "None", "Linear", "Geodesic", "Arc3Points", "Loxodromic", "Elliptical", "Conic", "CircularArcCenterPointWithRadiusEnd".

Remarks

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

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

50-9.1.1.5 Curve **CreateCurve**(Point *startPoint*, Point *endPoint*, CurveSegment[] *segments*)

Return Value

Curve

A Lua table containing a curve object.

Parameters

startPoint: Point

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

endPoint: Point

End point for the curve. Host creates by calling *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 curve directly; the curve is intended to be passed from the host back to the scripting catalogue.

50-9.1.1.6 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 composite curve directly; the composite curve is intended to be passed from the host back to the scripting catalogue.

50-9.1.1.7 Surface **CreateSurface**(SpatialAssociation *exteriorRing*, SpatialAssociation[] *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[]

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 surface directly; the surface is intended to be passed from the host back to the scripting catalogue.

50-9.1.2 Miscellaneous Functions

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

50-9.1.2.1 string **ConvertToJSON**(Table *data*)

Return Value

UTF-8 encoded string containing the JSON representation of *data*.

Parameters

data: Table

A Lua table that is to be converted to JSON.

Remarks

Converts the given Lua table into its UTF-8 encoded string representation using JavaScript Object Notation (JSON). Note that scripting catalogues do not accept JSON formatted data; this routine is a convenience for hosts which would rather parse JSON as opposed to Lua tables.

The following table shows the mapping of types between Lua and JSON:

Lua Type	JSON Type	Comments
Table	object	Each string indexed field in the Lua table is represented using a name/value pair in the JSON object. Tables containing integer indices (Lua arrays) map to JSON array values where the name maps to "" (empty string). The sequence of values are equivalent in the Lua and JSON arrays.
Array	array	
Number	number	
String	string	
Boolean	true/false	
Nil	null	

Example

The following shows a Lua table representing a feature instance along with the equivalent JSON representation.

Lua Table:

Feature		
Field	**Type**	**Value**
Type	string	Feature
ID	string	127
Code	string	beaconLateral
Colour	[integer]	1, 3
featureName	table	Name = beacon 1

JSON string:

```
{
        "Type" : "Feature"
        "ID" : "127"
        "Code" : "beaconLateral"
        "colour" : [1, 3]
        "featureName" : { "Name" : "beacon 1" }
}
```

50-9.2 Standard Host Functions

The host must provide a set of "callback" functions that provide the scripting environment with: access to the hosts representation of the S-100 General Data Model, access to type information for any entity defined by the General Data Model, and access to spatial operations which can be used to perform relational tests and operations on spatial elements defined by the General Data 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 Data 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.

50-9.2.1 General Data Model Access Functions

The host must implement the functions described on the following pages to allow the scripting environment to access the General Data Model of a dataset. These functions provide the scripting environment with access to features, spatials, attribute values, and information associations.

50-9.2.1.1 string[] **HostDatasetGetFeatureIDs**(string *datasetID*)

Return Value

string[]

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

Parameters

datasetID: string

Used by the host to uniquely identify a dataset.

Remarks

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

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

50-9.2.1.2 *string* **HostFeatureGetType**(string *featureID*)

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*.

50-9.2.1.3 *variant* **HostFeatureGetSimpleAttribute**(string *featureID*, path *path*, string *attributeCode*)

Return Value

nil

Attribute value is unknown.

string[]

The textual representation of each attribute value, as described in section 50-9.1. 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 section 50-7.1.

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.

50-9.2.1.4 *variant* **HostFeatureGetAttributeCount**(string *featureID*, path *path*, string *attributeCode*)

Return Value

nil

The requested attribute is not valid at the path for the feature instance.

integer

The number of matching attributes that exist at the path for the feature instance. If the multiplicity of the attribute is 1, then this value is -1.

Parameters

featureID: string

Used by the host to uniquely identify a feature instance.

path: path

An attribute path as described in section 50-7.1.

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. If the attribute is not valid at this path according to the feature catalogue, the host returns nil.

For valid attributes, the host returns the number of attributes at the requested path. This number can be zero. If the attribute is single valued (i.e. upper multiplicity of one) and present the host returns -1.

50-9.2.1.5 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.

50-9.2.1.6 *variant* **HostFeatureGetAssociatedFeatureIDs**(string *featureID*, string *associationCode*, string *roleCode*)

Return Value

nil

The feature association is not valid for this feature.

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

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 the feature 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* should be used for lookup.

50-9.2.1.7 *variant* **HostFeatureGetAssociatedInformationIDs**(string *featureID*, string *associationCode*, string *roleCode*)

Return Value

nil

The information association is not valid for this feature.

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

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 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.

50-9.2.1.8 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 creates the spatial using one of the following standard catalogue functions: *CreatePoint, CreateMultiPoint, CreateCurve, CreateCompositeCurve,* or *CreateSurface*.

50-9.2.1.9 *variant* **HostSpatialGetAssociatedInformationIDs**(string spatialID, string associationCode, string roleCode)

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

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.

50-9.2.1.10 string[] **HostSpatialGetAssociatedFeatureIDs**(string *spatialID*)

Return Value

string[]

A Lua array containing the requested associated feature IDs for 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.

50-9.2.1.11 *variant* **HostInformationGetSimpleAttribute**(string *informationID*, path *path*, string *attributeCode*)

Return Value

nil

Attribute value is unknown.

string[]

The textual representation of each attribute value, as described in section 50-9.1. An array is returned even if the attribute has a single value.

Parameters

informationID: string

Used by the host to uniquely identify an information instance.

path: path

An attribute path as defined in section 50-7.1.

attributeCode: string

One of the attribute codes defined in the feature catalogue for the information type identified by *informationID*.

Remarks

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

50-9.2.1.12 *variant* **HostInformationGetAttributeCount**(string *informationID*, path *path*, string *attributeCode*)

Return Value

nil

The requested attribute is not valid at the path for the information instance.

integer

The number of matching attributes that exist at the path for the information instance. If the multiplicity of the attribute is 1, then this value is -1.

Parameters

informationID: string

Used by the host to uniquely identify an information instance.

path: path

An attribute path as defined in section 50-7.1.

attributeCode: string

One of the attribute codes defined in the feature catalogue for the information type identified by *informationID*.

Remarks

Instructs the host to return the number of attributes matching *attributeCode* at the given attribute path for the given information instance. If the attribute is not valid at this path according to the feature catalogue, the host returns nil.

For valid attributes, the host returns the number of attributes at the requested path. This number can be zero. If the attribute is single valued (i.e. upper multiplicity of one) and present the host returns -1.

50-9.2.2 Type Information Access Functions

These functions allow the scripting environment to query the type information for any entity defined by General Data Model for any loaded datasets. The type information provided must be consistent with that provided by the feature catalogue for the relevant product.

The host must implement the functions described on the following pages to provide the scripting environment access to type information for entities defined by the General Data Model.

> ***NOTE: Type information access functions are not included in this draft. They were not needed to support portrayal; the proof-of-concept for scripting. Type information access functions will be required for data validation, and to support future yet to be defined scripting domains.***

50-9.2.3 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.

50-9.2.3.1 boolean **HostSpatialRelate**(string *spatialID1*, string *spatialID2*, string *intersectionPatternMatrix*)

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. E.g. 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)*.

50-9.2.4 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.

50-9.2.4.1 void **HostDebuggerEntry**(string *debugAction*, string *message*)

Return Value

None

Parameters

debugAction: string

Indicates the requested debugger action:

break – Pause execution of the script.

trace – Display a string in the debugging console.

start_profiler – Begin line by line profiling of the script code.

stop_profiler – Stop line by line profiling of the script code.

message: string

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.