GIRA

SDK logic node Documentation

Last updated: 26.01.2021

Contents

1 About this documentation

This documentation introduces the developer to the Software Development Kit (SDK) logic node, enabling them to develop logic nodes for the Logic Editor in the Gira Project Assistant (GPA). An enclosed Visual Studio project, in which the logic nodes are programmed in C#, is used for this purpose.

1.1 Target group

This documentation is aimed at people who already have the following knowledge:

- Developing in the programming language C#
- A basic understanding of the JSON format
- Using the Visual Studio 2017 development environment
- Using the Gira Project Assistant
    - How data points work
    - Using the Logic Editor

2 Basics

2.1 Logic nodes

Logic pages can be created using the Logic Editor in the Gira Project Assistant (GPA). These logic pages are visual representations of logic circuits that are executed on the device (e.g. Gira X1 or Gira L1) once the GPA project has been started up. Logic pages are made up of logic nodes and the connections between them.

A logic node adopts values via one or more inputs, processes them according to its internal programming and then outputs new calculated values at one or more outputs. The values at the outputs are transferred to the inputs of another logic node via a connection, which is represented on the logic page by a line between an input and an output. This enables creation of even complex logic circuits. Up until now, the logic has only communicated with the rest of the project via the *Input* and *Output* logic nodes, whose values are linked to random data points. Other communication channels can be developed with the SDK.

The GPA already contains some logic nodes by default, such as the *OR gate*, the *PID controller* or the *comparator*. The SDK logic node enables the developer to create their own logic nodes and use them in logic pages. New logic nodes are very useful for special applications. This SDK logic node already contains some examples, such as *Aggregation.* To name but one example, this node can be used to calculate the mean value of the values present at the inputs. For this application, it is very helpful to have your own logic node, because it is very time-consuming to calculate the mean value by linking other logic nodes and because the newly created logic node can be reused in many projects.

3 Using the examples

As part of the SDK logic node, a Visual Studio 2017 Solution is available in the *Example* and *Template* folders. The Solution in the *Example* folder contains a sample project named *ExampleNodes*, which implements several different nodes. The *Template* folder also contains a prepared Solution named *LogicNodes*, which can be used as a template for new logic nodes. There is also a unit test project for each of the two projects.

The unit tests use the *NUnit* framework, which is downloaded with the *NuGet* package manager integrated in Visual Studio. The tests are run with Visual Studio's Test Explorer. The *LogicNodesTest* project only contains an empty test case that is available to the developer to write their own tests for the logic node.

If the *ExampleNodes* project has been built successfully, the *LogicNodesSDK.Logic.Examples-1.0.0.zip* file that contains the sample nodes is located in the Solution's *Zip* sub-folder. The logic nodes in the .zip file are not yet cryptographically signed. This means that they can indeed be imported into the Gira Project Assistant (GPA) and tested in the simulation, but unsigned logic nodes cannot be transferred to the executing Gira device.

The *Add Logic Nodes* button is used in the Logic Editor tile to install the nodes in the GPA.

A certificate must be applied for to begin with for signing purposes. The complete process is described in section 4 Signing logic nodes below.

4 Signing logic nodes

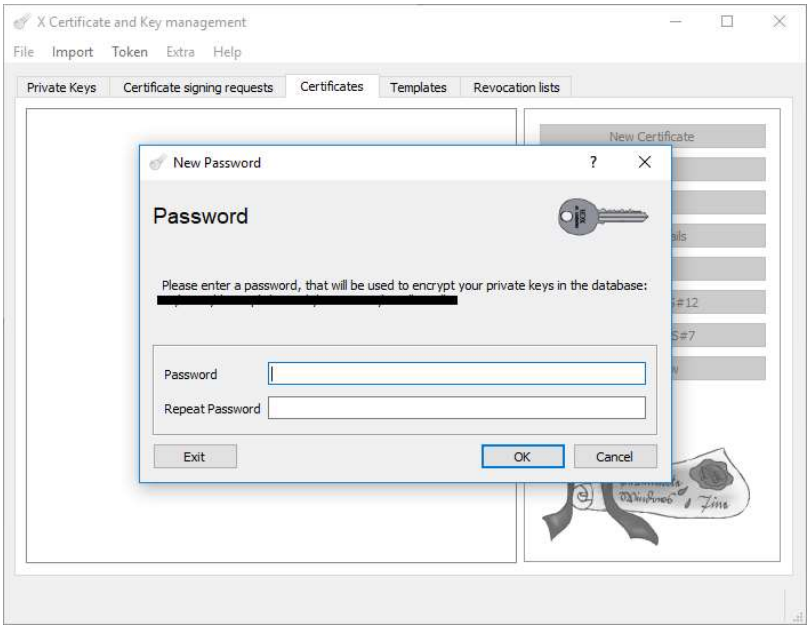
Logic nodes are signed with the *SignLogicNodes* program contained in the SDK. This program requires a PKCS#12 certificate issued by Gira. The developer must submit a certificate signing request to receive a certificate.

4.1 Creating a certificate signing request

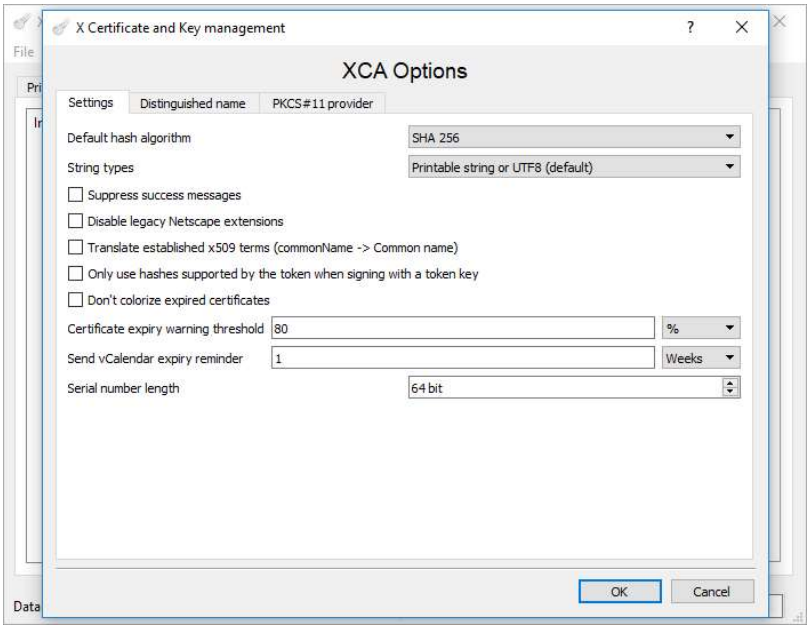
It is advisable to use the *X Certificate and Key Management* program to create the certificate signing request. The program is available at https://hohnstaedt.de/xca/. It must be installed and launched. The menu navigation is described in this document using the English language setting.

When certificates or keys are created for the first time, a database must be created for the program. Select the *File -> New DataBase* menu item to do so. Alternatively, an existing database can be imported with *File -> Open DataBase*.

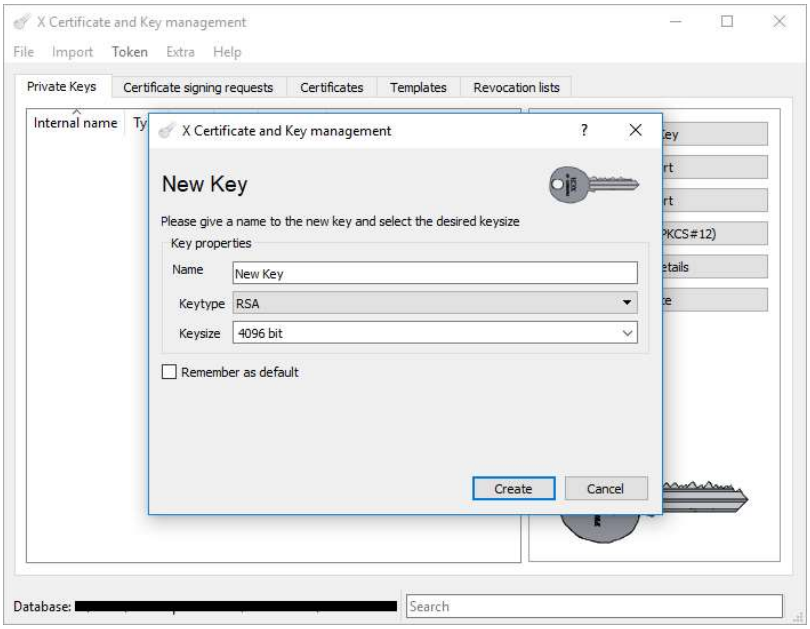
Each database is password-protected and can be used independently of the operating system. This password is set when the database is created.



First of all, it must be ensured that the *default hash algorithm* is set to *SHA 256* and that this setting is made if necessary. This setting can be found under the *File -> Options* menu item.



A private key must be generated. To do this, use the *New Key* button on the *Private Keys* tab and specify a random name for the key. *RSA* and *4096 bit* must be selected in the *Keytype* and *Keysize* drop-down menus.



A signing request is made with the private key. The *Certificate signing requests* tab must be selected for this purpose. Click on the *New Request* button to open a new window. In this window, the developer's personal data must be entered on the *Subject* tab:

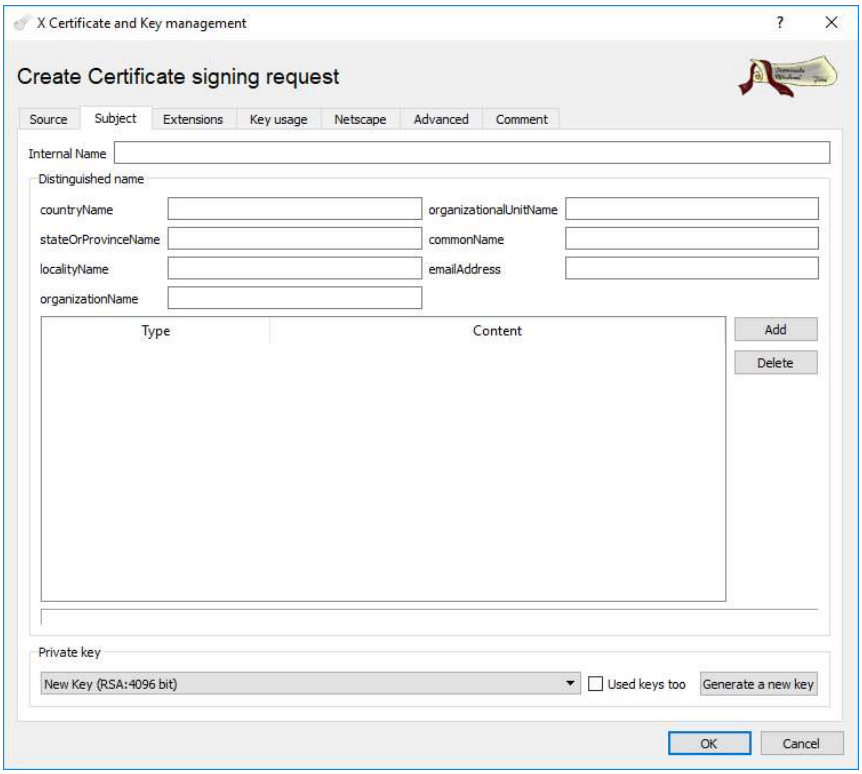
Internal Name	The certificate signing request's display name
countryName	The location's country code (e.g. DE for Germany)
localityName	The location's town / city name
organizationName	Company name (left blank for private individuals)
commonName	The applicant's name
emailAddress	The applicant's e-mail address

If the certificate is to be issued for a private individual, the *organizationName* field must be left blank.

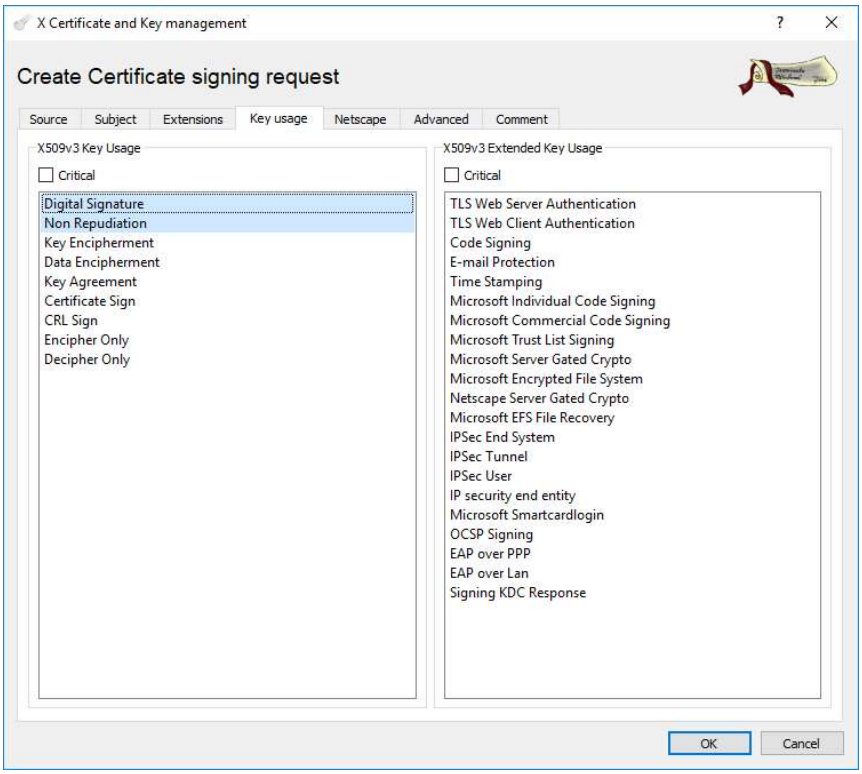
The data entered here is intended for display in the GPA and on the device website, so users can contact the publisher of these logic nodes directly if they have any problems.

Your personal data will be collected and processed based on the privacy policy attached to this document.

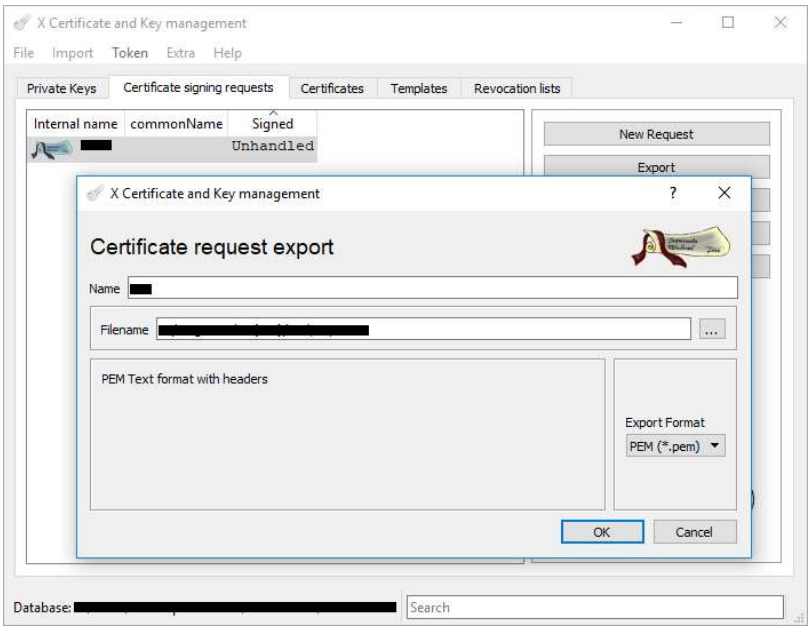
A private key must be selected in the *Private key* drop-down menu; the key just created can be selected here.



On the *Key usage* tab, *Digital Signature* and *Non Repudiation* must be selected by clicking on them. With this information, the certificate signing request is complete and can be generated by clicking on the *OK* button.



The *Export* button is used to export this certificate signing request. The *PEM* format must be selected under *Export Format*. This *.pem* file must be sent to *developer@gira.de* by e-mail. *Your personal data will be collected and processed based on the privacy policy attached to this document.* Your certificate signing request will then be processed and you should receive an answer from Gira containing your signed certificate after some time.



NOTE

Please never send your X Certificate database, your private key or one of your passwords to Gira. Do not share these files or this information with anyone who is not supposed to be able to publish logic nodes on your behalf.

4.2 Importing the created certificate from Gira

As soon as the certificate signing request has been processed, the requester receives a certificate signed by Gira which is imported into the *X Certificate and Key Management* program. To do this, a dialogue in which the certificate received must be selected is opened on the *Certificates* tab with the *Import PKCS#7* button. The user is asked in a window that opens which of the certificate chain's certificates are to be imported. Since all the certificates have to be imported, the *Import All* button is selected here.

In this imported certificate chain, the certificate previously created and now signed by Gira must be exported. The entire certificate chain is expanded on the *Certificates* tab and the bottom-most certificate selected for this purpose. Click on the *Export* button to export the certificate. In the file dialogue that opens, ***PKCS#12 chain*** *(*.p12)* must be selected as the file format.

The certificate signed by Gira is valid for 10 years by default. However, the validity period of the logic nodes' certificates is not checked on the devices, so the logic nodes remain functional even after the validity period has elapsed.

Once these 10 years have passed, a new certificate request can be submitted to be able to sign new logic nodes with a current certificate.

4.3 Using the certificate to sign logic nodes

The logic nodes are signed using the *SignLogicNodes* program contained in the SDK.

The program must be called up from the command line interface with three parameters: the path to a signed certificate, this certificate's password and the path to the logic node *.zip* file to be signed.

SignLogicNodes.exe <p12 Certificate> <Password> <Path to the logic node .zip file>

Alternatively, this step can also be entered in the project as a *post-build event*.

As described in section 3 Using the examples, the signed node is added to the GPA using the *Add Logic Nodes* button. This means that the logic nodes can be used in the GPA and also uploaded to a Gira X1 or a Gira L1 as a project component.

5 Description of the API

5.1 Foreword

The SDK logic node provides a programming interface, known as an 'API' for short, for developing logic nodes. The use of this logic node API (hereinafter only referred to as the 'API') ensures that the developed logic nodes are compatible with the GPA and the other logic nodes.

5.2 Creating a new logic node class

Each newly developed logic node is implemented as a class in C#.

```
using LogicModule.Nodes.Helpers;
using LogicModule.ObjectModel;

public class Node : ILogicNode
{
  public Node(INodeContext context)
  {
    ...
  }
  ...
}
```

The *Node* class inherits from the *ILogicNode* class, which exists in the *LogicModule.ObjectModel* namespace. The constructor of the *Node* class must be implemented in such a way that it is transferred a variable of the *INodeContext* type. The *INodeContext* type also exists in the *LogicModule.ObjectModel* namespace.

ILogicNode is an interface class, so all methods of this class (*Execute*, *Startup*, *Localize* and *Validate*) must be implemented in the *Node* class. More information about these methods can be found in section 5.4. The *LogicNodeBase* class implements useful dummies for these methods. So it makes sense, especially for beginners, for methods to inherit from the *LogicNodeBase* node. It is located in the *LogicModule.Nodes.Helpers* namespace.

The *context* variable is the access point to the API. Via this interface, the node requests special services to access the various functions such as data points, times or input and output generation. The available services are described in section 5.3 below.

5.3 API services

The API provides services for the different functionalities. Instances of these services are called up from the *INodeContext* object in the constructor. To do this, the line

```
‘ServiceName’ typeService = context.GetService<‘ServiceName’>();
```

must be called up. The *‘ServiceName’* placeholder is the type of the desired service. The following services are available in the API.

5.3.1 ITypeService

The *ITypeService* provides methods to define the properties of the logic node’s ports. A port is an input, an output or a parameter. Ports are represented by *ValueObjects*. The *ITypeService* provides methods for generating specific *ValueObjects* instances. The section below lists the methods used to generate different types of instances.

```
AnyValueObject CreateAny(
  string typeName, string name, object defaultValue = null);

BoolValueObject CreateBool(
  string typeName, string name, bool? defaultValue = null);

ByteValueObject CreateByte(
  string typeName, string name, byte? defaultValue = null, string unit = null);

IntValueObject CreateInt(
  string typeName, string name, int? defaultValue = null, string unit = null);

DoubleValueObject CreateDouble(
  string typeName, string name, double? defaultValue = null, string unit = null);

UIntValueObject CreateUInt(
  string typeName, string name, uint? defaultValue = null, string unit = null);

UShortValueObject CreateUShort(
  string typeName, string name, ushort? defaultValue = null, string unit = null);

TimeSpanValueObject CreateTimeSpan(
 string typeName, string name, TimeSpan? defaultValue = null, string unit = null);

DateTimeValueObject CreateDateTime(
 string typeName, string name, DateTime? defaultValue = null, string unit = null);

StringValueObject CreateString(
  string typeName, string name, string defaultValue = null);
```

These methods are structured according to a schema:

- The first parameter is the name of the type to be generated. It defines the port's data type. An example is 'NUMBER' to define a port that adopts or transfers a value of the *Number* type.
- The second parameter is the name of the port or *ValueObject*. This is also displayed in the GPA as a *port name*, unless it is overwritten by a translation into another language; also see section 5.5. The *port name* must be unique within a logic node.
- The third parameter is always optional and is the initial value that the port has when generating the node, i.e. when adding it to the logic page. However, before the logic is started, the GPA user can change or even delete the port's value. Ports whose values are unavailable or have been deleted are read with the value *null*.
- The fourth parameter is only available for some methods and is always optional too. It indicates in which physical unit the port's numerical value is to be interpreted, e.g. whether a length is given in metres, centimetres or feet. It is only used to inform the node user and is displayed by the GPA as a port property.

The types exist in the *LogicModule.ObjectModel.TypeSystem* namespace.

In addition to the above functions, *ITypeService* also provides the *CreateValueObject* and *CreateEnum* methods.

```
IValueObject CreateValueObject(
  string typeName, string name, object defaultValue = null);

EnumValueObject CreateEnum(
  string typeName, string name, string[] allowedValues, string defaultValue=null);
```

These methods are similar to the others in the list of generating functions. Most of their parameters will be used as already described above. However, they still require a separate explanation.

The *CreateValueObject* method generates an instance of the *IValueObject* type. The *IValueObject* is an interface type and cannot be used directly; *IValueObject* must be converted into one of the other *ValueObject* types by a cast. This enables replacement of all the other functions from the list of functions to be generated. However, it is advisable to use the more specific functions to save the cast.

The *CreateEnum* method defines a new *ValueObject* type. This type's possible values are strings from a list. The *typeName* parameter specifies the name of the new type so that further instances of this enum type can be created later on. The possible values that the enum type can adopt are transferred as an array of strings with *allowedValues*.

If another instance of a previously defined enum type is to be generated, *CreateEnum* offers an overload.

```
EnumValueObject CreateEnum(
  string typeName, string name, string defaultValue = null);
```

Only the array with the possible values is missing here. The *typeName* parameter must be the name of a previously defined enum type.

The *ColorConverter* sample logic node uses a parameter of the enum type to select the two possible conversion directions.

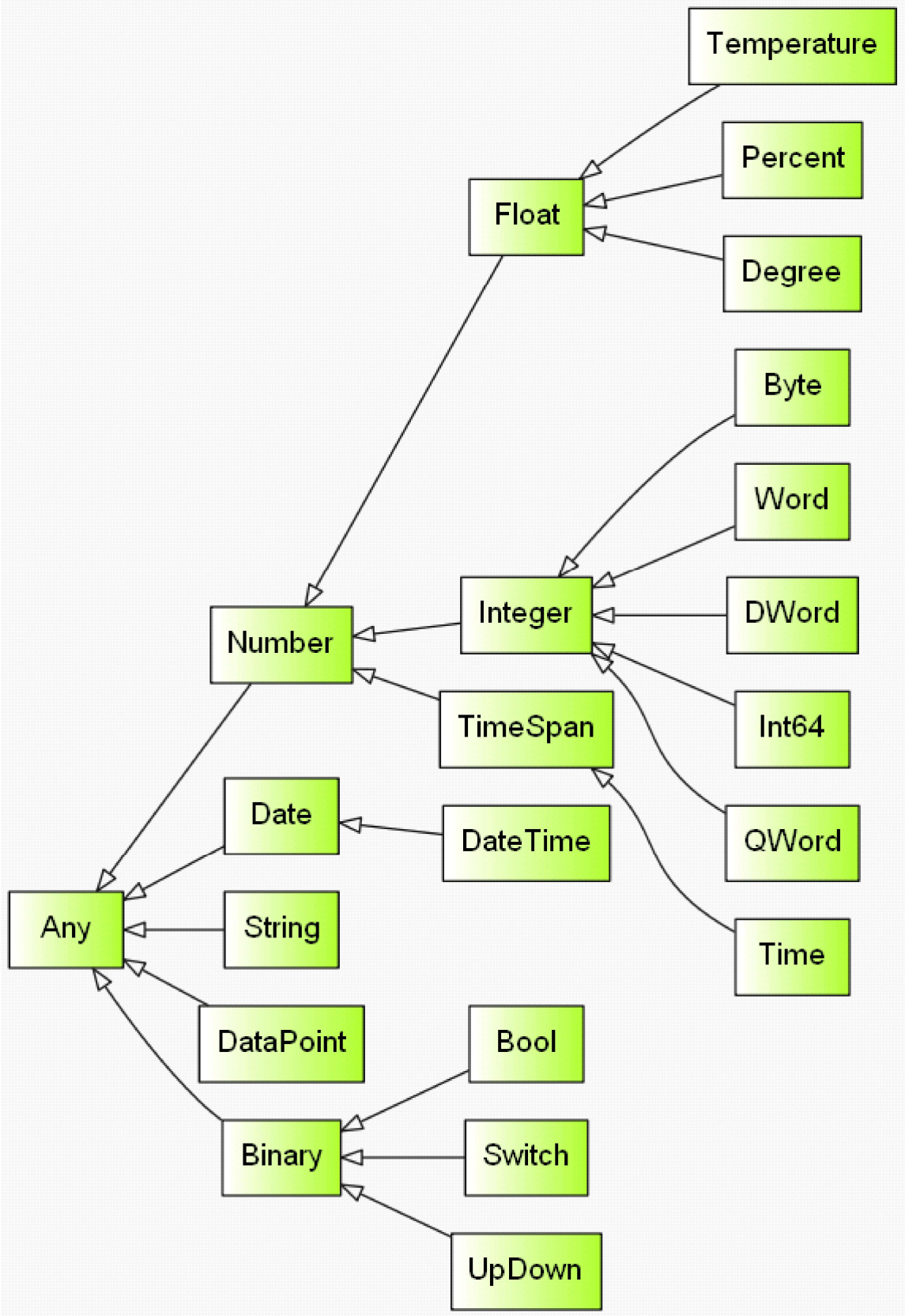
The complete port definition is described in section 5.3.1b. The type system is described here first of all.

5.3.1a The type system

The type system provides various data types for meaningful port typing. This prevents different nodes' ports whose connection makes no sense from being connected. For example, connecting an output of the 'STRING' type to an input of the 'NUMBER' type does not make sense. On the other hand, you should avoid having to implement a separate node for each type to keep the number of nodes and the development costs low. It is therefore advisable to select the type of ports as generally as possible, but as specifically as necessary.

An example of this is a logic node that adds the values of two or more inputs and outputs the result at the output. For this application, it makes sense to select 'NUMBER' for the ports' types. This ensures that the ports can be connected to ports of other nodes, e.g. of the *Integer*, *Float* or *Byte* type. Another conceivable example is angle addition. A logic node such as this adds the values at the inputs, applies a modulo 360 operation to the result and writes the result to the output. For this node, it is very useful to assign the 'DEGREE' type to the ports. This prevents the possibility of accidentally connecting ports to other ports that are incompatible with the 'DEGREE' type.

All of the implemented types and their relationships to one another are displayed in the illustration of the type system.



5.3.1b Complete port definition

A port is defined is in two steps. First of all, a property is created in the logic node's class.

```
[Input(DisplayOrder = 1, IsRequired = true)]
public DoubleValueObject Input { get; private set; }
```

The property must be *public*, but the *setter* can be set to *private*. The developer is free to select the property's name. The property's type must match the desired port type. In the example, a *DoubleValueObject* is defined so that the 'NUMBER' port type can be selected in the second step. The *Input* attribute indicates that an input is defined here. There are three attributes in total for ports:

- *Input* for defining an input
- *Output* for defining an output
- *Parameter* for defining a parameter

At the input property example, you can see that parameters can be transferred to the attribute. The following parameters are available.

Port parameter

DisplayOrder

DisplayOrder is available for all attributes (Input, Output, Parameter). The value of this parameter influences the display order in the GPA. The smaller *DisplayOrder*'s value, the further up the corresponding port is displayed in the visual representation in the GPA's Logic Editor. In the example below, *Output1* is displayed before *Output2*.

```
[Output(DisplayOrder = 2)]
public IValueObject Output2 { get; set;}


[Output(DisplayOrder = 1)]
public IValueObject Output1 { get; set;}
```

The following should be noted:

- If *DisplayOrder* was not set, the order is undefined.
- *DisplayOrder* applies to the *Output* attribute separately from the other two attributes, so outputs in the GPA are still always displayed to the right of the node, and inputs and parameters are always displayed to the left.
- The value of *DisplayOrder* must be greater than or equal to 1; all other values are interpreted as if the value was not set.

InitOrder

The *InitOrder* parameter is available for the *Input* and *Parameter* attributes. This influences the order of initialisation. In the example below, the *Datapoint* parameter is initialised first, then the *Trigger* input. If *InitOrder* is not set, the order is undefined.

```
[Input(InitOrder = 2)]
public AnyValueObject Trigger { get; set; }

[Parameter(InitOrder = 1)]
public DataPointValueObject Datapoint { get; private set; }
```

Here, it should again be noted that the lowest possible value is 1. All values less than 1 are interpreted as not set.

AsTitle

This parameter is only available for the Parameter attribute. This is a flag, so only the values *true* or *false* can be assigned. If this flag is set to *true*, this indicates that the parameter is fundamentally important for the node. This flag can be set to *true* for a maximum of one parameter per node. If a node such as this is added to the logic page, the GPA expects a value for this parameter to be entered immediately. If the value is missing, the corresponding text box is highlighted in yellow by the GPA. The value entered for this port is used as the node's title. This function is used in the 'Input' and 'Output' logic nodes to prompt the user to enter a data point whose name is then displayed as the node's title.

```
[Parameter(AsTitle = true)]
public DataPointValueObject Datapoint { get; private set; }
```

IsDefaultShown

All three attributes offer the *IsDefaultShown* parameter. This is also a flag. It determines whether the port is displayed by default or hidden. In the case of a parameter, this determines whether the value of the parameter is displayed on the node by default. The default setting for this flag is *true*, so each port is displayed when the node is generated, unless the flag was actively set to *false*.

```
[Input]
public IValueObject Input1 { get; private set; }

[Input(IsDefaultShown = true)]
public IValueObject Input2 { get; private set; }

[Input(IsDefaultShown = false)]
public IValueObject Input3 { get; private set; }
```

The two inputs *Input1* and *Input2* are displayed on the node by default in the GPA, whereas *Input3* is not displayed by default. Regardless of this flag, the GPA user can show or hide the ports. This flag specifies only the default setting that the port has when the logic node is added to the logic page.

IsRequired

The *IsRequired* parameter is also available for all three attributes, but has a slightly different effect on the various attributes. This parameter is again a flag and can therefore only adopt the values *true* or *false*; the default setting is *false*. If the flag's value is set to *true*, the port can no longer be hidden in the GPA for all three attributes. If *IsDefaultShown* = *false* is also specified, this is overwritten with the *IsRequired* = *true* setting and the port is displayed by default. For inputs, it is possible to switch between two states in the GPA's Logic Editor. Either the input receives its value from another node's output, or the user must manually enter a fixed value. For inputs with *IsRequired* = *true*, this switching option is omitted and the input must always obtain its value from another node's output.

```
[Input(IsRequired = true)]
public IValueObject Input { get; set; }
```

IsInput

This parameter is only available for the Input attribute, so only for inputs and also only for situations where *IsRequired = true* was not set. This is again a flag with the default setting *true*. It specifies how the input is displayed to the user, whether a fixed value must be entered by default or whether the input must be connected to an output. This flag only affects the default display when the node is added to the logic page. It is also possible for the user to configure the node and switch between the two types of value assignment.

```
[Input]
public IValueObject Input1 { get; private set; }

[Input(IsInput = false)]
public IValueObject Input2 { get; private set; }

[Input(IsInput = true)]
public IValueObject Input3 { get; private set; }
```

The *Input2* input is displayed in the GPA when the node is generated in such a way that the user must enter a fixed value, whereas *Input1* and *Input3* are displayed in such a way that a connection to an output must be established.

Instantiation

If all desired port parameters have been assigned values, the next step of port definition follows in the node constructor. The properties of the ports must be instantiated by assigning them an instance created by the *ITypeService* object. How this works is best illustrated by an example.

```
public class Node : LogicNodeBase
{
  ...
  [Input(DisplayOrder = 2, IsDefaultShown = false, IsInput = true)]
  public BoolValueObject Reset { get; private set; }
  ...

  public Node(INodeContext context)
  : base(context)
  {
    ...
    ITypeService typeService = context.GetService<ITypeService>();
    this.Reset = typeService.CreateBool(PortTypes.Binary, "Reset", false);
    ...
  }
}
```

In the constructor of the *Node* class, the *Reset* input is assigned the display name *Reset* and the initial value *false*. The port type is also set to ‘BINARY’. All available port type names are present in the static *PortTypes* class as constant string variables, so the developer has a guide for the notation.

For some of the type system’s types, the instances have meta information such as a minimum or maximum value. These are set and called up after instantiation. For *ValueObjects* of the ‘NUMBER’ type and the derived types, there is a minimum and maximum value that can be specified if required.

```
this.Count = typeService.CreateInt(PortTypes.Integer, "Count", 1);
this.Count.MinValue = 1;
this.Count.MaxValue = 10;
```

The set minimum and maximum values are displayed by the GPA as port properties. A maximum permissible length can be specified for *StringValueObjects*; this is set using the *MaxLength* parameter.

```
this.InputStr = typeService.CreateString(PortTypes.String, "Input String", "42");
this.InputStr.MaxLength = 12;
```

This meta information is also displayed by the GPA as a port property. Some port types have implicit minimum and maximum values. For example, the value of a 'BYTE'-type port is always between 0 and 255. It can be specified how the logic node behaves if a value that is outside the permissible range is written to a port. There are two possible settings:

- Saturation: The value is automatically adjusted to fall within the valid value range. If, for example, the value exceeds the set or implicit maximum value, it is replaced by the maximum value which the port then adopts.
- Exception: The wrong value leads to an exception in the logic. This means that the port does not adopt any value changes. Additionally, the error is recorded and the part of the logic in which the node is located is stopped.

The default behaviour is that described under 'Saturation'. This meta information is set by the line

```
this.Count.OutOfRangeBehavior = OutOfRangeBehavior.Saturation;
```

List of ports

For many logic nodes, the number of inputs or outputs needs to be set dynamically while the logic page is being created. To name but one example, this functionality is used in the *Aggregation* sample node.

To generate ports dynamically, a list of inputs or outputs is defined as a property of the node class. The syntax is very similar to the definition of a single input or output.

```
[Input]
public List<DoubleValueObject> Inputs { get; private set; }
```

This list must be initialised in the node class constructor.

```
this.Inputs = new List<DoubleValueObject>();
```

Two help functions are provided to fill the list with the corresponding port's instances. These are located in the *LogicModule.Nodes.Helpers* namespace within the *ListHelpers* class.

```
void UpdateListLength<T>(
IList<T> list, int newCount, Func<int, IValueObject> creator,
EventHandler<ValueChangedEventArgs> handler = null)
where T : IValueObject;
```

```
void ConnectListToCounter<T>(
IList<T> list, ValueObject<int> countParameter, Func<int, IValueObject> creator,
EventHandler<ValueChangedEventArgs> handler, Action listLengthChangedCallback =
null)
where T : IValueObject;
```

The *UpdateListLength* function fills the list of inputs or outputs with a fixed number of instances. The first parameter must be the list itself. The second parameter indicates the required list length. To fill the list, the function needs a rule to generate new *IValueObject* instances; this is the third parameter, *creator*. How this rule is defined is described below. There is also an optional fourth parameter: the event handler, *handler*. This is assigned to the *ValueSet* event (see below under ValueSet), which is called up when the list length is changed, is added to the newly created ports or is removed from the deleted ones. This has the effect that the method represented by *handler* is called up as soon as one of the values of the ports in the list changes.

A common application is for the GPA user to manually specify the required number of ports. This is also the case with the *Aggregation* sample node. A parameter, Port, is available to the user so they can enter the desired number of ports. With the *ConnectListToCounter* method, the value of the parameter is automatically linked to the list length. The first parameter of this function is the dynamic list of ports; the second is the parameter Port, whose value specifies the list length. Note that the port type of this parameter must be *Integer*. The smallest permissible value for the list length is '0', so the values that the parameter can adopt must be limited to positive values by the properties *MinValue* and *MaxValue*. The third and fourth parameters of the function are analogous to those of *UpdateListLength*. The fifth parameter is an optional callback function that is called up each time the port list length is changed.

The *creator* parameter can be generated using the *ITypeService* object.

```
Func<int, IValueObject> creator =
this.typeService.GetValueObjectCreator(PortTypes.Number, InputPrefix);
```

The first parameter of *GetValueObjectCreator* is the port type of the ports generated with *creator*, and the second is a string that is used as the name prefix for the ports to be generated. For example: The developer chooses 'Input' for *InputPrefix* and transfers the *creator* variable thus created to the *UpdateListLength* or *ConnectListToCounter* function. These methods thus generate ports with the names 'Input 1', 'Input 2', 'Input 3', etc. Note that the numbering of the names starts with '1', but the list index of the first port is '0'.

5.3.2 ISchedulerService

The *ISchedulerService* interface provides methods to execute actions at a specific time. The current system time can also be queried. The interface provides an abstraction layer at the real system time. This abstraction makes it possible to simulate a different time, e.g. when

simulating a logic page in the GPA. Direct access to the system time via the *.NET Framework* is to be avoided in the node.

The *ISchedulerService* interface provides the following methods and properties.

```
SchedulerToken InvokeIn(TimeSpan delay, Action action);

SchedulerToken InvokeAt(DateTime dueTime, Action action);

bool Remove(SchedulerToken schedulerToken);

DateTime Now;
```

The *InvokeIn* method is used to set an *action* that is executed after the *delay* time. Similarly, the action transferred to the *InvokeAt* method is executed at the *dueTime* fixed time according to UTC. Both functions return a token of the *SchedulerToken* type. If necessary, this token can be stored in a variable that can be transferred to the 'Remove' method to remove the corresponding action from the queue of actions. The *SchedulerToken* type is a nullable type. The abstracted local system time can be queried with *Now*. Note: The *Now* method returns the local time, whereby *InvokeAt* expects a time specification according to UTC.

The following example shows how these methods interact.

```
public class Enterprise : LogicNodeBase
{
  ...
  private ISchedulerService schedulerService;
  private SchedulerToken abortToken;
  ...
  public Enterprise(INodeContext context)
    : base(context)
  {
    ...
    this.schedulerService = context.GetService<ISchedulerService>();
    TimeSpan countdown = new TimeSpan(0, 0, 10, 0);  //10min
    this.abortToken = this.schedulerService.InvokeIn(countdown, this.SelfDestruction);
    //Do not lose this token or let the Klingons find it
    ...
  }
  ...
  private void SelfDestruction()
  {
    ...  //Explosion
  }
  private void Abort()
  {
    if (abortToken != null)
    {
      this.schedulerService.Remove(this.abortToken);
      this.abortToken = null;
    }
  }
  ...
}
```

5.3.3 IPersistenceService

With this interface, the node stores values over the device's runtime. The following methods are available.

```
void SetValue(ILogicNode node, string key, string value);

string GetValue(ILogicNode node, string key);

void DeleteValue(ILogicNode node, string key);
```

All three methods require the instance of the calling logic node as the first parameter, so *this* must be transferred. The second parameter is the key used to access the value. The *SetValue* method stores or overwrites the *value* of the *node* and the *key*. The *GetValue* method calls the value stored by *SetValue* back up and outputs it as a return parameter. If no value is stored for *key*, *GetValue* returns*null*. The value is deleted again with the *DeleteValue* method. Values can only be stored and called up as strings.

The stored values can also be called up with *GetValue* even after the device on which the logic node is used has been restarted. A stored value is still available even if the GPA project in which the same logic node (with the same ID) is used is relaunched. The ID of the logic node is generated automatically as soon as the node is added to the logic page. A value can be uniquely assigned to a node and a *key* by means of the node ID. This prevents one node from accessing the values of another one. This also means that values stored via the *IPersistenceService* interface are lost as soon as the node is removed from the logic page.

The stored values are deleted or no longer available if

- The logic node is removed from the logic page.
- The Gira device is assigned to another project or the assignment is removed.
- Start-up is carried out with the 'Delete application data' flag set.
- The Gira device is reset to factory settings.

The *SetValue* and *GetValue* methods throw a *CommunicationException* if access to the values fails or takes too long.

5.3.4 IEditorService

The methods of this interface influence the logic page's editability and should be used with great care. Use of this interface only makes sense in the constructor and the event handlers that may be registered there, because it only affects the port types displayed in the GPA and thus only the operation during the creation of the logic page. There are a number of nodes in which a value is passed through without further evaluation under certain conditions. The *Send-By-Change* node is an example of this. For this node, no restriction may be made with regard to the port types, so the corresponding inputs and outputs use the 'ANY' port type. If the user has linked one of the ports to a port of another node, it is clear that both the input and the output of the *Send-By-Change* node must adopt the port type of this external port. The *IEditorService* offers the following methods for mapping such scenarios.

```
ISharedTypeToken RegisterSharedType(ILogicNode node, params IValueObject[] ports);

void ClearAllSharedTypeRegistration(ILogicNode node);
```

The *RegisterSharedType* method merges a group of ports that are transferred to the function listed into one group. If one of the ports in the group is connected to another port of a more

specific port type when the logic page is created (in the image of the type model, this means that a type that is arranged further 'right' than the actual port type is linked), all other ports in the group also adopt this type. The first parameter to be transferred to the method is the *this* reference to the calling node. The return parameter is a token that is used to dissolve the group again at a later time. The *ISharedTypeToken* type provides the *Unregister* method for this purpose.

```
void Unregister();
```

Once this method has been called up, transfer of the port type between the group's ports is cancelled again. The *ClearAllSharedTypeRegistration* method dissolves all created groups of ports again.

The example of the *Send-By-Change* node is particularly simple. In the constructor, only one line is called up to link the input's and the output's port types.

```
context.GetService<IEditorService>().RegisterSharedType(
this, this.Input, this.Output);
```

5.4 Functionality of the Programming node

This section describes how the functionality of the node is implemented. The *ILogicNode* base class, from which each node class inherits directly or indirectly, provides two methods that must be implemented by the node class. The following code should be considered as an example.

```
public class Node : ILogicNode
{
 public Node(INodeContext context)
 {
  ...
 }
 public override void Startup()
 {
  ... //This code is executed once when the logic engine starts up
 }
 public override void Execute()
 {
  ... //This code is executed every time a value is written to an input,
      //and all inputs have a value
 }
 ...
}
```

The *Startup* method is executed only once when the node is started. This method is very useful for testing whether the node's inputs or parameters have been assigned values during configuration and for reading these values. The *LogicNodeBase* class implements an empty *Startup* method. The *Execute* method is key for the node to work. It is called up each time a value (not necessarily a new value) is written to an input by linking to another node and all inputs have a valid value (not *null*). In the method, the values are called up and processed and,

if necessary, the outputs are updated. The *LogicNodeBase* class implements an empty *Execute* method here too.

5.4.1 Properties of *IValueObject*

WasSet

WasSet is a flag and can only be used meaningfully in the *Execute* method. This property is set to *true* when the value of the corresponding input is set. This allows the user to determine which input triggers calling-up of the *Execute* method. At this point, the user can react in different ways to the value changes at the various inputs. This is useful, for example, if the node has two inputs: a reset input that resets the node in a defined form when a value is received, and an input that triggers the node's actual function. Once the *Execute* method has been executed, the *WasSet* values of all inputs are automatically reset to *false*.

HasValue

HasValue is also a flag. This property can be used to query whether the value of this port is valid, i.e. not *null*. This can be the case for ports without an initial value or if the initial value has been removed by the user. It is therefore necessary to check for all inputs and outputs whether a value is available before calling up the value.

Value

This property is vitally important for the *IValueObject* instance. It contains the actual value. The data type is specified by the port type; for *BoolValueObject*, *Value* is always of the *bool* type, for *UIntValueObject*, *Value* is always of the *uint* type, and so on. For an *IValueObject* instance that is not specified in more detail and for *AnyValueObject* instances, the *Value* type is *object*. *Value* is both writeable and readable.

ValueEquals

The *ValueEquals* method is used to check whether the value stored in the *IValueObject* instance matches the transferred value. This is a short notation for value comparisons. *ValueEquals* checks internally whether the value is set at the *IValueObject* before the value comparison with *HasValue*. For the actual value comparison, *ValueEquals* uses the *Object.Equals* method or its overloads from the *.NET Framework*.

BlockGraph

This property is only useful for outputs and may only be used for them. *BlockGraph* sets the value of the output to *null* and blocks the execution of all downstream logic nodes. This is useful, for example, if the node is performing an asynchronous operation and is waiting for a return value from this operation. Execution of the downstream logic nodes is stopped during the wait period. Only when another value is written to the output for which *BlockGraph* was called up is the execution of the downstream logic continued with the newly written value.

NOTE: The simulation of a logic page in the GPA does not reproduce the behaviour of *BlockGraph* correctly.

ValueSet

This property is an event handler and available for all ports. A change in a port's value triggers an event that calls up all functions registered under *ValueSet*. The '+=' operator is used to register another function for this event.

```
this.Direction.ValueSet += this.DirectionOnValueSet;
```

This example is taken from the program code of the *ColorConverter* sample node. Here, the *DirectionOnValueSet* function is added to the *ValueSet* event for the *Direction* parameter. Each time the parameter's value is set, the *DirectionOnValueSet* function is automatically called up. The *ValueSet* property is particularly useful if, as for the *ColorConverter* node, inputs and outputs are to be generated dynamically.

5.5 Checking the logic nodes

The *ILogicNode* interface class also contains the *Validate* method, which must also be overwritten in the node class. The *Validate* method is not intended to be called up by the node; rather, it is only called up by the GPA. It is used to inform the GPA whether the node has been meaningfully initialised. The validation of the logic nodes can be called up in the GPA at various points, e.g. by the user starting the simulation of a logic page. Each time the simulation of the logic page is started, the GPA calls up all nodes' *Validate* methods. If a logic node reports an error, starting of the simulation is aborted and an error message is displayed in the GPA. The logic pages are also validated when the user begins the project start-up or manually triggers testing of the project. For these two cases, details in the information window show the user which logic node the validation failed for and which error message the node generates. This functionality is used, for example, in the *Threshold with Hysteresis* logic node to ensure that the lower threshold is lower than the upper one.

```
public override ValidationResult Validate(string language)
{
  if ((this.LowerThreshold.HasValue && this.UpperThreshold.HasValue) &&
       this.LowerThreshold.Value > this.UpperThreshold.Value)
  {
    return new ValidationResult { HasError = true, Message =
           this.Localize(language,"HysteresisValidationErrorMessage") };
  }
  return base.Validate(language);
}
```

The method has a *ValidationResult* object as return parameter. *ValidationResult* has the properties *HasError* (a flag indicating whether there is a validation error; *false* means there is no error) and *Message*, which is of the *string* type. If the *HasError* flag is *true*, *Message* is displayed to the user in the information window. The *language* transfer parameter is a language code that the GPA uses to query the *Message* error message in a particular language for the method (details on translations are provided in section 5.6). The implementation of this method in *LogicNodeBase* returns the fixed *HasError=false* and an empty *Message*. The validation is therefore always successful for this class.

5.6 Translation of the logic nodes

The logic nodes can be made available to the user in different languages. The *Localize* method of the *ILogicNode* interface class must be implemented in the node class for this purpose. This method is called up by the GPA to translate any port name present in the node.

```
string Localize(string language, string key);
```

The structure is very simple: *key* is the string to be translated and *language* is the code of the language that the content of *key* is to be translated into. The *Localize* return value is the translated result that is displayed by the GPA as the port name. The *Localize* method is also used inside the node in the example from section 5.5 to translate the *Message* error message, for example. The GPA transfers the language code of the language set in the GPA to the *Localize* method. The following languages are available in the GPA.

Language codes according to ISO 639-1	Language
de	German
en	English
es	Spanish
nl	Dutch
ru	Russian
zh	Chinese
(it)	(Italian)

Italian is not available yet (GPA v3.1), but will be available in later versions.

The fact that the *LogicNodeBase* class implements the *Localize* method is very convenient. The following section describes how to use the *Localize* method implementation in the *LogicNodeBase* and *LocalizablePrefixLogicNodeBase* classes.

The *LogicNodeBase Localize* method uses the *ResourceManager* from the *System.Resources* namespace. This class is documented in the *.NET Framework*. *.resx* files are used as translation tables. *.resx* files can be conveniently created and edited with Visual Studio. The files must exist in the same directory as the Visual Studio project file. A *.resx* file must be created for each language, as must a default *.resx* file if necessary. If there is only one resources file (including multiple languages), this file will be used for translation. If multiple files are found, the *Resources.<Language Code>.resx* file is preferably used, otherwise a random one is used. <Language Code> corresponds to the language code of the language provided by this translation file. The file name of the default *.resx* file does not contain a language code, i.e. <name>.resx.

Visual Studio displays the *.resx* files in the editor as a table. The strings to be translated are entered in the first column (*Name*); the corresponding translation is entered in the second column (*Value*). In case the language requested with the *language* parameter does not have a translation file or the corresponding *key* has not been translated, the *Localize* method uses the default *.resx* file. It is advisable to create a default translation file such as this, because it will make the port names displayed more user-friendly. In the source code, the developer may under certain circumstances enter abbreviations for the port names or avoid spaces and special characters. The port names are subsequently adapted with the default translation. If neither a language-specific nor a default translation was found, this *Localize* method returns the *key* input value.

The translation is slightly more complex if the Ports in the Form of a List node is defined. Especially for this case, the *LocalizablePrefixLogicNodeBase* class is available in the *LogicModule.Nodes.Helpers* namespace. This class inherits from *LogicNodeBase* and only overwrites the *Localize* method. It may be useful for the developer to have the node class inherited from *LocalizablePrefixLogicNodeBase*. The constructor of this class requires two transfer parameters.

```
public LocalizablePrefixLogicNodeBase(INodeContext context, string inputPrefix)
```

The *context* parameter must be transferred from the node class to this base class; *inputPrefix* is the prefix of the port names from the port list as also transferred to the *GetValueObjectCreator* method. The *Localize* method handles strings (*key*) starting with the prefix separately and replaces only the prefix with the translation of this prefix given in the *.resx* file. The *Localize* method also inserts a space between the prefix and the rest of the string if the translation of the prefix does not end with a space. All other strings (*key*) that do not begin with the *inputPrefix* prefix are treated in the same way as the *Localize* method of the *LogicNodeBase* base class. The developer must ensure that the port names and the prefix are selected so that no other port name unintentionally begins with the prefix.

It is also possible for a node to contain more than one port list, e.g. one for inputs and one for outputs. These port lists have different prefixes. In this case, the SDK does not provide a pre-implemented *Localize* method, so the developer must implement it themselves.

6 The Manifest.json file

Each logic node library needs a *Manifest.json* file in which the metadata of the containing logic nodes is defined. This file must exist in the same folder as the sources of the logic nodes themselves. The SDK logic node contains two *Manifest.json* files: one for the *ExampleNodes* project and one for the *LogicNodes* project. The respective Visual Studio projects already contain these *Manifest.json* files, so they can be easily edited from within Visual Studio.

6.1 Adjusting the coding

The *Manifest.json* file must be saved with the *UTF-8-BOM* coding so that special characters such as umlauts are correctly displayed in the names and descriptions of the logic nodes in the GPA.

In Visual Studio, the coding can be adjusted at:

File -> Save <File Name> As.... In the File Save dialogue, the *Save with Encoding...* item must be selected in the drop-down menu next to *Save*. In the *Advanced Save Options* window that opens, the *Unicode (UTF-8 with signature) – Codepage 650001* option must be selected under *Encoding*.

6.2 Sample Manifest.json file


```
{
  "PackageFormatVersion": "1.0",
  "Assembly": "LogicNodesSDK.Logic.Examples.dll",
  "PackageName": {
    "en": "Example logic nodes",
    "de": "Beispiellogikbausteine"
  },
  "DependentFiles": [
    "LogicModule.Nodes.Helpers.dll"
  ],
  "Version": "1.0.0",
  "Author": "Gira Giersiepen GmbH & Co. KG",
  "Copyright": "Gira Giersiepen GmbH & Co. KG (copyright)",
  "DeveloperId": "LogicNodesSDK",
  "License": "Free",
  "PackageId": "183FBF6C-AE53-4393-A2A5-2CBE0F1696AF",
  "Nodes": [
    {
      "Type": "LogicNodesSDK.Logic.Examples.BasicNode",
      "Name": {
        "en": "Examples: Basic logic node",
        "de": "Beispiele: Einfacher Logikbaustein"
      },
      "IsConverter": false,
      "Category": "Node",
      "DefaultIcon": "icons/GenericLogicNode.png",
      "HelpTooltip": {
        "en": "Basic logic node that sets the output to the same value as the input.",
        "de": "Einfacher Logikbaustein, der den Ausgang auf den gleichen Wert wie den
Eingang setzt."
      },
      "HelpFileReference": null
    },
    {
      "Type": "LogicNodesSDK.Logic.Examples.Aggregation",
      "Name": {
        "en": "Examples: Aggregation",
        "de": "Beispiele: Aggregation"
      },
      "IsConverter": false,
      "Category": "Node",
      "DefaultIcon": "icons/GenericLogicNode.png",
      "HelpTooltip": {
        "en": "Calls aggregate functions for multiple inputs: minimum, maximum, sum,
average.",
        "de": "Ruft Aggregatfunktionen für mehrere Eingänge auf: Minimum, Maximum,
Summe, Durchschnitt."
      },
      "HelpFileReference": null
    }
  ]
}
```

6.3 Description of the entries in the Manifest.json file

Key	Sample value	Application
PackageFormatVersion	'1.0'	The version of the Manifest.json format. Always '*1.0*'.
Assembly	'LogicNodesSDK.Logic.Examples.dll'	File name of the logic node library or the name of the DLL.
PackageName	{ "en": "Example logic nodes", "de": "Beispiellogik-bausteine" }	A JSON object whose keys are country codes and whose values are the name of the node library in the respective language.
DependentFiles	['LogicModule.Nodes.Helpers.dll']	JSON array with file names of the required DLLs.
Version	'1.0.0'	Version of the logic node. Only the highest version of a logic node is available in the GPA. Displayed in the GPA in the logic node's properties under *Version*.
Author	'Gira Giersiepen GmbH & Co. KG'	The developer of the logic node.
Copyright	''	Field is not evaluated.
DeveloperId	'LogicNodesSDK'	The developer's ID. The ID is assigned with the certificate.
License	'Free'	The library's licence model. Possible values are: 'Free' or 'Device'.
PackageId	'183FBF6C-AE53-4393-A2A5-2CBE0F1696AF'	The library's GUID. Here, you have to generate your own. In Visual Studio under *Tools -> Create GUID*.
Nodes	[<See extra table>]	JSON array with one object per logic node. The specific structure is described in the table below.

Structure of objects in the nodes array

Key	Sample value	Application
Type	'LogicNodesSDK.Logic.Examples.ColorConverter'	The node's class name, including the namespaces.
Name	{ "en": "Color converter", "de": "Farben-Konverter" }	A JSON object whose keys are country codes and whose values are the name of the node in the respective language.
IsConverter	*false*	Indicates whether the node can be connected directly downstream of the output when an output is right-clicked.
Category	'Node'	*Node* should always be entered for this version of the SDK logic node.
DefaultIcon	'icons/GenericLogicNode.png'	Relative path to the logic node's icon.
HelpTooltip	{ "en": "Converts colors between 1x 3 byte and 3x 1 byte.", "de": "Wandelt Farbwerte zwischen 1x 3 Byte und 3x 1 Byte um." }	A JSON object whose keys are country codes and whose values are a short description of how the logic node works in the respective language.
HelpFileReference	'BasicArithmetic'	Help file that opens in the GPA when you click on 'More Information'. The value is treated differently depending on where the entered value begins. Details are given in section 6.4.

6.4 Licence model

The *License* entry, which indicates the licence the node package should be available under, exists in the *Manifest.json* file. The table already shows that there are two possible values for this entry.

6.4.1 Free

With the 'Free' licence model, nodes from the package can be used on any device as often as required. No additional licence file is required on the device. Additionally, 'Free' is the default value that will be used if the *License* entry in the *Manifest.json* file is not present.

6.4.2 Device

The other alternative is the 'Device' licence model. Logic node packages can be distributed via the Gira AppShop for a fee. To use these nodes, a licence file that binds the node package to a fixed device via the MAC address is also assigned. This licence file can either be imported manually into the GPA or downloaded automatically from the device when the internet connection is active.

Licence files with limited validity can be issued to customers, so a package logic node is no longer executable on the device after this validity has expired. Trial versions of a logic node package that cease to be valid after 30 days, for example, can be distributed in this way.

In the Gira Smart Home App, a user is notified 30 days in advance if a licence will expire in the near future.

6.5 Description of the *HelpFileReference* in the Manifest.json file

The GPA can display a help page in the browser for each logic node using the default browser set in the user's system. The help page is called up when '…More information' is clicked on in the node's information window. The *HelpFileReference* entry in the *Manifest.json* file specifies how the GPA locates and calls up the help page.

The <Value> placeholder here always stands for the value of the *HelpFileReference* entry in the *Manifest.json* file. <Value> always starts with a control character that specifies how the help page is found by the GPA.

Control character '!'

The control character '!' instructs the GPA to call up an external URL. The complete URL, including the protocol name, must follow directly after the '!' for this purpose.

```
"HelpFileReference": "!https://gira.de/",
```

This way of providing a help page is very easy to implement and allows the developer to customise the help page even without the node being supplied in a new version. However, the user needs an active internet connection.

Control character '$'

The node can also provide a *.html* file that is displayed in the browser. <Value> must start with the control character '$', followed by the file name, for this purpose.

```
"HelpFileReference": "$help.html",
```

The *help.html* file must be present in the *$(TargetDir)* target directory before Visual Studio builds the node. The *LogicNodeTool.exe* program, which is called up in the *post-build event*, generates the node's *.zip* file, including the *help.html* file.

Control character '@'

For some nodes, it makes sense for the help page to be displayed in the language set in the GPA. To implement this, <Value> must start with the control character '@'. As before, the corresponding *.html* file must already exist in the target directory before construction.

The exact path is *$(TargetDir)\Help*\<Language Code>\<Assembly>*.html*. This path structure is retained if the *LogicNodeTool.exe* program generates the node's *.zip* file. <Language Code> is one of the language codes from the table, whereby the GPA uses the language set internally. <Assembly> is the full file name of the logic node library, as specified in the *Manifest.json* file after the *Assembly* key (without the *.dll*extension). After the control character '@', <Value> specifies the ID of a jump label in the *.html* file.

```
"HelpFileReference": "@node",
```

When the help page is called up, the GPA opens the corresponding <Assembly>*.html* and

jumps to the *node* jump label. It is therefore possible to place several logic nodes in one help file and navigate to the correct position with the jump label.

7 Hints and tips

Before a logic node is published, the risk of errors must be minimised; of course, they can never be excluded entirely. The following items are a list of errors and problems that often occur.

Debugging / testing logic nodes

To test the functionality of the created logic nodes, it is advisable to use the prepared NUnit Test project by writing sufficient test cases there to ensure the node's functionality.

It is also possible to debug in the simulation of the GPA. In Visual Studio, you can use the *Attach to Process* feature to do this. Visual Studio must attach to the GPA process. If, in the GPA's logic node simulation, a breakpoint is hit by its node, Visual Studio stops the GPA process, then debugging can take place.

Versioning

The logic nodes are added to the GPA using the *Add Logic Nodes* button. However, the node is only adopted by the GPA if the added version of the node does not already exist in the GPA. In other words, every time the node's source code is changed, the version number in the *Manifest.json* file must be increased before the node is built, otherwise the node cannot be added to the GPA. The new version number must be entered manually in the *Manifest.json* file under the *Version* key.

The latest version of the node is only used for nodes that are newly added to a logic page. If an older version of the logic node already exists on the logic page, it retains its older version. The *Update Logic Nodes* button updates the version of the nodes that are already used on a logic page to the latest available version. In any case, a situation where there are several versions of the same node in one logic page must be avoided. If a logic page such as this is loaded to a Gira device, this leads to the logic behaving unexpectedly.

Before a node is published, it must therefore be checked whether the version number is incremented correctly.

Translation and help

When translating the logic nodes, it is advisable to use the existing *Localize* methods from the *LogicNodeBase* or *LocalizablePrefixLogicNodeBase* classes if possible. It is also advisable to generate a default *.resx* file to display meaningful port names in the GPA for the user. If a port list is used, care must be taken to ensure that the name prefix is not used at the same time at the start of another port name to avoid incorrect translations. Additionally, the port names assigned to the ports during instantiation must be unique within the node.

The developer must bear a few things in mind when creating language-specific help files. There is no default help file to use if the language set in the GPA is not available. A help file should therefore be created for each language available in the GPA, even if it is not written in the corresponding language. A help file in another language is more useful for the user than no help file at all.

Frequent misuse

Accessing non-existent values:

Before accessing a port's value, *HasValue* must be used to check whether this value exists. Many operations called up with *null* values cause the logic to crash. It also needs to be checked whether the value exists if the port has been assigned an initial value in the constructor, since the GPA user can delete the port's initial value.

Exiting the value range for list lengths:

A node often needs to use a parameter for the GPA user to set the number of inputs or outputs. The *ConnectListToCounter* help function expects a parameter of the 'INTEGER' port type. It is therefore absolutely essential that the parameter's permissible values with the *MinValue* and *MaxValue* properties are limited to positive values that are not 'too big'. Otherwise, the user can enter a value that causes crashing.

Different time zones for *ISchedulerService*:

Actions that are executed at a fixed time can be set with the *InvokeAt* method of the *ISchedulerService* interface. The time must be entered according to the UTC time standard. If the fixed time is calculated from a time span and the current time, it is important to note that the *Now* method returns the local time and not the time according to UTC.

.NET version

It is absolutely essential that the '.NET Framework 4.5' or '.NET Framework 4.0' *target framework* is selected in the project settings for the logic node. Otherwise, the logic node on the device cannot be executed correctly. This is already set in all projects delivered with the system.

Supported libraries on the devices

The LogicEngine on the Gira X1 / Gira L1 runs mono under Linux.

For the Gira X1, this is mono 4.0.2, and for the Gira L1, mono 3.0.1. Both versions do not have all libraries from the .NET Framework available.

List of available mono libraries:

- Mono.Security
- System
- System.Configuration
- System.Core
- System.Data
- System.Drawing (*)
- System.Numerics
- System.Runtime.Remoting (*)
- System.Runtime.Serialization
- System.Security

- System.ServiceModel
- System.ServiceModel.Web
- System.Transactions
- System.Web
- System.Xml
- System.Xml.Linq

(*) = not available on the L1

If the developer would like to use other external libraries, they must be copied to the device together with the node in the zip file.