# PLCopen and OPC Foundation:

# OPC UA Information Model

# for IEC 61131-3

# -

# Release 1.00

# March 24, 2010

# CONTENTS

**FIGURES**

**TABLES**

# PLCOPEN / OPC FOUNDATION
_____

## AGREEMENT OF USE

This Agreement shall be governed by and construed under the laws of the Netherlands.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

## 1   Scope

This specification was created by a joint working group of the OPC Foundation and PLCopen. It defines an OPC UA *Information Model* to represent the IEC 61131-3 architectural models.

It is important that the controller as a main component of automation systems is accessible in the vertical information integration which will be strongly influenced by OPC UA. OPC UA servers which represent their underlying manufacturer specific controllers in a similar, IEC 61131-3 based manner provide a substantial advantage for client applications as e.g. visualizations or MES. Controller vendors may reduce costs for the development of these OPC UA servers if an OPC UA Information Model for IEC 61131-3 is used.

OPC Foundation

The OPC Foundation defines standards for online data exchange between automation systems. They address access to current data (OPC DA), alarms and events (OPC A&E) and historical data (OPC HDA). Those standards are successfully applied in industrial automation.

The new OPC Unified Architecture (OPC UA) unifies the existing standards and brings them to state-of-the-art technology using service-oriented architecture (SOA). Platform-independent technology allows the deployment of OPC UA beyond current OPC applications only running on Windows-based PC systems. OPC UA can also run on embedded systems as well as Linux / UNIX based enterprise systems. The provided information can be generically modelled and therefore arbitrary information models can be provided using OPC UA.

PLCopen

PLCopen, as an organization active in industrial control, is creating a higher efficiency in your application software development: in one-off projects as well as in higher volume products. As such it is based on standard available tools to which extensions are and will be defined.

With results like Motion Control Library, Safety, XML specification, Reusability Level and Conformity Level, PLCopen made solid contributions to the community, extending the hardware independence from the software code, as well as reusability of the code and coupling to external software tools. One of the core activities of PLCopen is focused around IEC 61131-3, the only global standard for industrial control programming. It harmonizes the way people design and operate industrial controls by standardizing the programming interface. This allows people with different backgrounds and skills to create different elements of a program during different stages of the software lifecycle: specification, design, implementation, testing, installation and maintenance. Yet all pieces adhere to a common structure and work together harmoniously.

## 2   Reference documents

IEC 61131-3: IEC 61131-3, 2nd Edition, Programmable Controllers – Part 3: Programming Languages

PLCopen XML: XML Formats for IEC 61131-3 Version 2.0

UA Part 1: OPC UA Part 1 – Concepts, Version 1.01 or later

UA Part 3: OPC UA Part 3 – Address Space Model, Version 1.01 or later

UA Part 4: OPC UA Part 4 – Services, Version 1.01 or later

UA Part 5: OPC UA Part 5 – Information Model, Version 1.01 or later

UA Part 7: OPC UA Part 7 – Profiles, Version 1.00 or later

UA Part 8: OPC UA Part 8 – Data Access, Version 1.01 or later

UA Part DI: OPC UA Companion Specification For Devices, Version 1.00 or later

## 3    Terms, definitions, and conventions

### 3.1    Use of terms

Defined terms of  and OPC UA specifications, types and their components defined in OPC UA specifications and in this specification are highlighted with italic in this document.

### 3.2    IEC 61131-3

The following terms defined in IEC 61131-3 apply.

1)    Body
2)    Configuration
3)    Function
4)    Function Block
5)    Program
6)    Program Organization Unit
7)    Resource
8)    Task
9)    Variable

To avoid naming conflicts with OPC UA terms the prefix Ctrl for controller is used together with these terms like *Ctrl Variable* or *Ctrl Program*.

### 3.3    OPC UA Part 1 terms

The following terms defined in UA Part 1 apply.

1)    AddressSpace
2)    Attribute
3)    Event
4)    Information Model
5)    Method
6)    MonitoredItem
7)    Node
8)    NodeClass
9)    Notification
10)    Object
11)    ObjectType
12)    Profile
13)    Reference
14)    ReferenceType
15)    Service
16)    Service Set
17)    Subscription
18)    Variable
19)    View

### 3.4  OPC UA Part 3 terms

The following terms defined in UA Part 3 apply.

1) DataVariable
2) EventType
3) Hierarchical Reference
4) InstanceDeclaration
5) ModellingRule
6) Property
7) SourceNode
8) TargetNode
9) TypeDefinitionNode
10) VariableType

### 3.5    OPC UA DI terms

The following terms defined in UA Part DI apply.

1) Block
2) Device
3) Parameter

### 3.6    IEC 61131-3 UA Information Model terms

#### 3.6.1  Controller

A controller is a digitally operating electronic system, designed for use in an industrial environment, which uses a programmable memory for the internal storage of user-oriented instructions for implementing specific functions such as logic, sequencing, timing, counting and arithmetic, to control, through digital or analogue inputs and outputs, various types of machines or processes.

### 3.7    Abbreviations and symbols

A&E         Alarms & Events
Ctrl        Controller
DA          Data Access
HDA         Historical Data Access
HMI         Human-Machine Interface
IEC         International Electrotechnical Commission
MES         Manufacturing Execution System
PLC         Programmable Logic Controller
SCADA       Supervisory Control And Data Acquisition
UA          Unified Architecture
XML         Extensible Markup Language

### 3.8    Fundamentals

#### 3.8.1  Introduction to IEC 61131-3

IEC 61131-3 is the first real endeavour to standardize programming languages for industrial automation. With its worldwide support, it is independent of any single company.

IEC 61131-3 is the third part of the IEC 61131 family. This consists of: (1) General information, (2) Equipment requirements and tests, (3) Programming languages, (4) User guidelines, (5) Communications, (6) Safety, (7) Fuzzy control programming, and (8) Guidelines for the application and implementation of programming languages.

IEC 61131-3   basically describes the Common Elements and Programming Languages.

### 3.8.1.1  Common Elements

### 3.8.1.1.1    Data Typing

Within the common elements, the data types are defined. Data typing prevents errors in an early stage. It is used to define the type of any parameter used. This avoids for instance dividing a Date by an Integer.

Common data types are Boolean, Integer, Real and Byte and Word, but also Date, Time_of_Day and String. Based on these, one can define own personal data types, known as derived data types. In this way one can define an analogue input channel as data type, and re-use this over an over again.

### 3.8.1.1.2    Ctrl Variables

*Ctrl Variables* are only assigned to explicit hardware addresses (e.g. input and outputs) in *Ctrl Configurations*, *Ctrl Resources* or *Ctrl Programs*. In this way a high level of hardware independency is created, supporting the reusability of the software.

The scopes of the *Ctrl Variables* are normally limited to the organization unit in which they are declared, e.g. local. This means that their names can be reused in other parts without any conflict, eliminating another source of errors. If the *Ctrl Variables* should have global scope, they have to be declared as such (VAR_GLOBAL). *Ctrl Variables* can be assigned an initial value at start up and cold restart, in order to have the right setting.

### 3.8.1.1.3    Ctrl Configuration, Ctrl Resources and Ctrl Tasks

These elements are integrated within the software model as defined in the standard (see below).
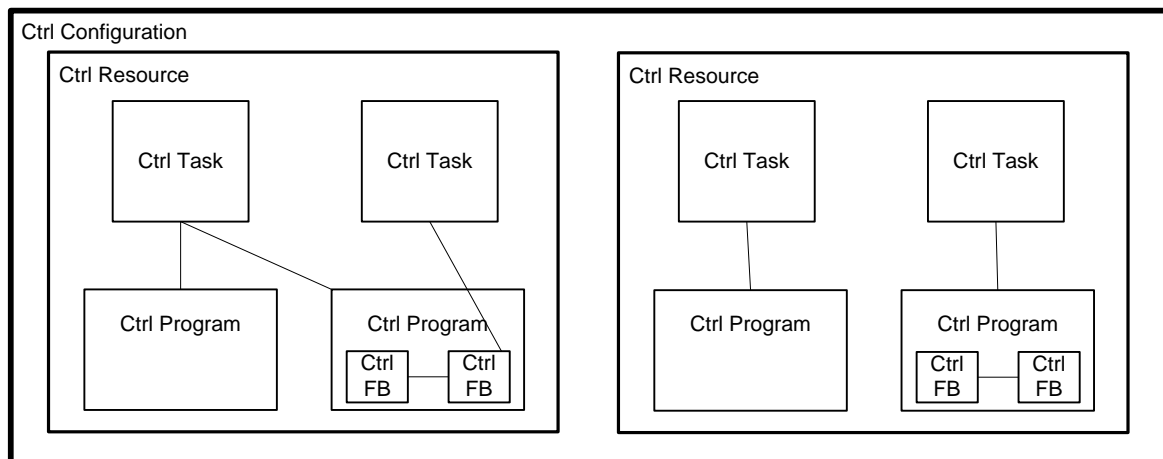


**Figure 1 – Software Model**

At the highest level, the entire software required to solve a particular control problem can be formulated as a *Ctrl Configuration*. A *Ctrl Configuration* is specific to a particular type of control system, including the arrangement of the hardware, i.e. processing resources, memory addresses for I/O channels and system capabilities.

Within a *Ctrl Configuration* one can define one or more *Ctrl Resources*. One can look at a *Ctrl Resource* as a processing facility that is able to execute  *Ctrl Programs*.

Within a *Ctrl Resource*, one or more *Ctrl Tasks* can be defined. *Ctrl Tasks* control the execution of a set of *Ctrl Programs* and/or *Ctrl Function Blocks*. These can either be executed

periodically or upon the occurrence of a specified trigger, such as the change of a *Ctrl Variable*.

*Ctrl Programs* are built from a number of different software elements written in any of the defined programming languages. Typically, a *Ctrl Program* consists of a network of *Ctrl Function*s and *Ctrl Function Blocks*, which are able to exchange data. *Ctrl Functions* and *Ctrl Function Blocks* are the basic building blocks, containing a data structure and an algorithm.

### 3.8.1.1.4    Ctrl Program Organization Units

Within IEC 61131-3, the *Ctrl Programs*, *Ctrl Function Blocks* and *Ctrl Functions* are called *Ctrl Program Organization Units*, POUs.

### 3.8.1.1.4.1    Ctrl Functions

IEC 61131-3 has defined standard *Ctrl Functions* and user defined *Ctrl Functions*. Standard *Ctrl Functions* are for instance ADD(ition), ABS(absolute), SQRT, SINus and COSinus. User defined *Ctrl Functions*, once defined, can be used over and over again.

### 3.8.1.1.5    Ctrl Function Blocks

*Ctrl Function Blocks* are the equivalent to integrated circuits, representing a specialized control function. They contain data as well as the algorithm, so they can keep track of the past (which is one of the differences w.r.t. *Ctrl Functions*). They have a well-defined interface and hidden internals, like an integrated circuit or black box. In this way they give a clear separation between different levels of programmers, or maintenance people.

A temperature control loop, or PID, is an excellent example of a *Ctrl Function Block*. Once defined, it can be used over and over again, in the same *Ctrl Program*, different *Ctrl Programs*, or even different projects. This makes them highly re-usable.

*Ctrl Function Blocks* can be written in any of the  languages, and in most cases even in "C". It this way they can be defined by the user. Derived *Ctrl Function Blocks* are based on the standard defined *Ctrl Function Blocks*, but also completely new, customized *Ctrl Function Blocks* are possible within the standard: it just provides the framework.

The interfaces of *Ctrl Functions* and *Ctrl Function Blocks* are described in the same way.

### 3.8.1.1.6    Sequential Function Chart

Within the standard Sequential Function Chart (SFC) is defined as a structuring tool. This means that syntax and semantics have been defined, leaving no room for dialects. The language consists of a textual and a graphical version.

### 3.8.1.1.7    Ctrl Programs

A *Ctrl Program* is a network of *Ctrl Functions* and *Ctrl Function Blocks*. A *Ctrl Program* can be written in any of the defined programming languages.

### 3.8.1.2  Programming Languages

Within the standard four programming languages are defined. This means that their syntax and semantics have been defined, leaving no room for dialects. The languages consist of textual and graphical versions:

- Instruction List, IL (textual)

- Structured Text, ST (textual)

- Ladder Diagram, LD (graphical)

- Function Block Diagram, FBD (graphical)

### 3.8.2  Introduction to OPC Unified Architecture

### 3.8.2.1  General

The main use case for OPC standards is the online data exchange between devices and HMI or SCADA systems using Data Access functionality. In this use case the device data is provided by an OPC server and is consumed by an OPC client integrated into the HMI or SCADA system. OPC DA provides functionality to browse through a hierarchical namespaces containing data items and to read, write and to monitor these items for data changes. The classic OPC standards are based on Microsoft COM/DCOM technology for the communication between software components from different vendors. Therefore classic OPC server and clients are restricted to Windows PC based automation systems.

OPC UA incorporates all features of classic OPC standards like OPC DA, A&E and HAD but defines platform independent communication mechanisms and generic, extensible and object-oriented modelling capabilities for the information a system wants to expose.

The OPC UA network communication part defines different mechanisms optimized for different use cases. The first version of OPC UA is defining an optimized binary TCP protocol for high performance intranet communication as well as a mapping to accepted internet standards like Web Services. The abstract communication model does not depend on a specific protocol mapping and allows adding new protocols in the future. Features like security, access control and reliability are directly built into the transport mechanisms. Based on the platform independence of the protocols, OPC UA servers and clients can be directly integrated into devices and controllers.

The OPC UA *Information Model* provides a standard way for *Servers* to expose *Objects* to *Clients*. *Objects* in OPC UA terms are composed of other *Objects*, *Variables* and *Methods.* OPC UA also allows relationships to other *Objects* to be expressed.

The set of *Objects* and related information that an OPC UA *Server* makes available to *Clients* is referred to as its *AddressSpace*. The elements of the OPC UA *Object* Model are represented in the *AddressSpace* as a set of *Nodes* described by *Attributes* and interconnected by *References*. OPC UA defines eight classes of *Nodes* to represent *AddressSpace* components. The classes are *Object*, *Variable*, *Method*, *ObjectType*, *DataType, ReferenceType* and *View*. Each *NodeClass* has a defined set of *Attributes*.

This specification makes use of two essential OPC UA *NodeClasses*: *Objects* and *Variables*.

*Objects* are used to represent components IEC 61131-3 software model like *Ctrl Program*, *Ctrl Task*, *Ctrl Resource* and *Ctrl Function Blocks*. An *Object* is associated to a corresponding *ObjectType* that provides definitions for that *Object*.

*Variables* are used to represent values. Two categories of *Variables* are defined, *Properties* and *DataVariables*.

*Properties* are *Server*-defined characteristics of *Objects*, *DataVariables* and other *Nodes*. *Properties* are not allowed to have *Properties* defined for them. An example for *Properties* of *Objects* is the *Priority Property* of a Ctrl *Task*.

*DataVariables* represent the contents of an *Object*. *DataVariables* may have component *DataVariables*. This is typically used by *Servers* to expose individual elements of arrays and structures. This specification uses *DataVariables* to represent data like the *Ctrl Variables* contained in Ctrl Programs* or in *Ctrl Function Blocks*.

### 3.8.2.2 Graphical Notation

OPC UA defines a graphical notation for an OPC UA *AddressSpace*. It defines graphical symbols for all *NodeClasses* and how different types of *References* between *Nodes* can be visualized. Figure 2 shows the symbols for the six *NodeClasses* used in this specification. *NodeClasses* representing types always have a shadow.
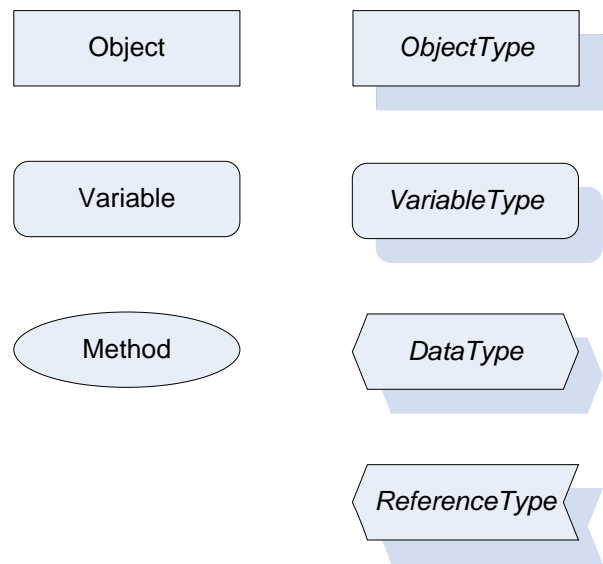


**Figure 2 – OPC UA *Graphical Notation for NodeClasses***

Figure 3 shows the symbols for the *ReferenceTypes* used in this specification. The *Reference* symbol is normally pointing from the source *Node* to the target *Node*. The only exception is the *HasSubType Reference*. The most important *References* like *HasComponent*, *HasProperty*, HasTypeDef*i*nition and *HasSubType* have special symbols avoiding the name of the *Reference*. For other *ReferenceTypes* or derived *ReferenceTypes* the name of the *ReferenceType* is used together with the symbol.
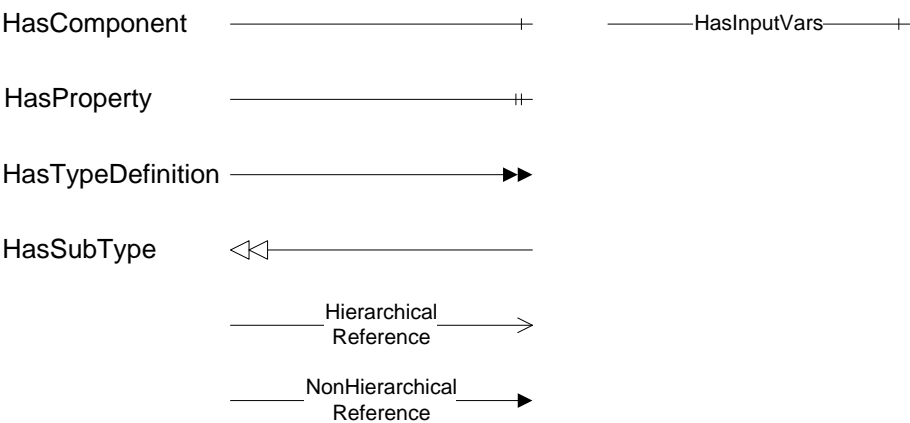


**Figure 3 – OPC UA *Graphical Notation for References***

Figure 4 shows a typical example for the use of the graphical notation. Object_A and Object_B are instances of the ObjectType_Y indicated by the HasTypeDefinition *References*. The ObjectType_Y is derived from ObjectType_X indicated by the HasSubType *Reference*. The Object_A has the components Variable_1, Variable_2 and Method_1.

To describe the components of an *Object* on the *ObjectType* the same *NodeClasses* and *References* are used on the *Object* and on the *ObjectType* like for ObjectType_Y in the

example. The instance *Nodes* used to describe an *ObjectType* are instance declaration *Nodes*.

To provide more detailed information for a *Node*, a subset or all *Attributes* and their values can be added to a graphical symbol.
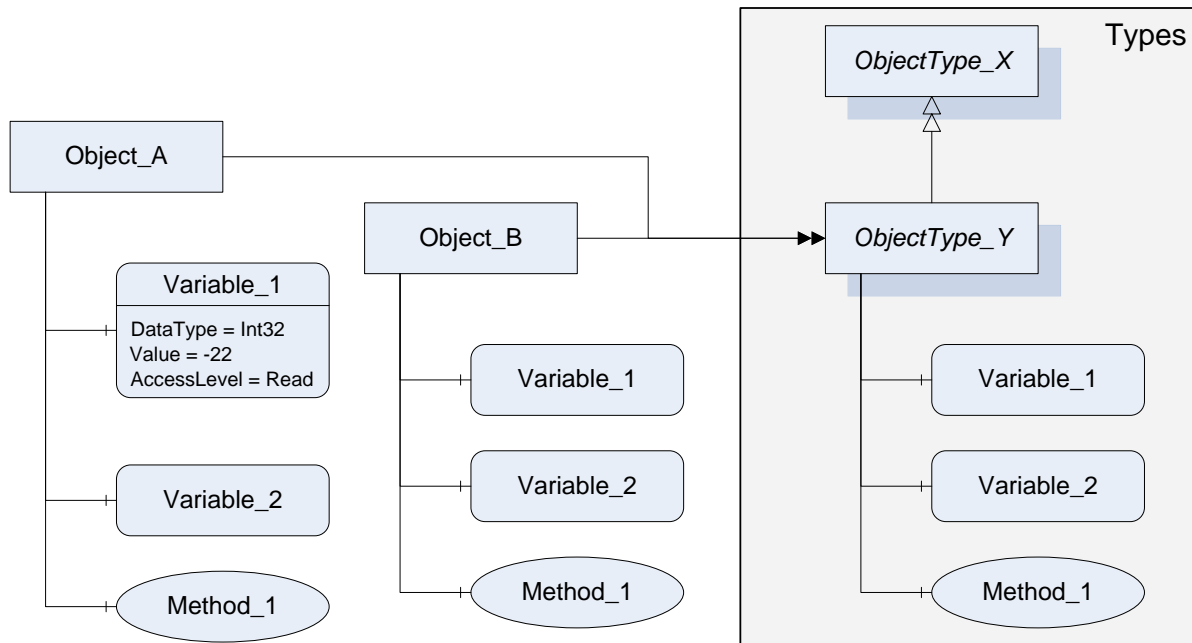


**Figure 4 – OPC UA *Graphical Notation Example***

### 3.8.2.3  Introduction to OPC UA Devices

The UA Part DI specification is an extension of the overall OPC Unified Architecture specification series and defines the information model associated with *Devices*. The model is intended to provide a unified view of *Devices* irrespective of the underlying *Device* protocols. Controllers are physical or logical *Devices* and the *Devices* model is therefore used as base for the IEC 61131-3 information model.

The *Devices* information model specifies different *ObjectTypes* and procedures used to represent *Devices* and related components like the communication infrastructure in an OPC UA *Address Space*. The main use cases are *Device* configuration and diagnostic but it allows a general and standardized way for any kind of application to access *Device* related information. The following examples illustrate the concepts used in this specification. See UA Part DI for the complete definition of the *Devices* information model.

Figure 5 shows an example for a temperature controller represented as *Device Object*. The component *ParameterSet* contains all *Variables* describing the *Device*. The component *MethodSet* contains all *Methods* provide by the *Device*. Both components are inherited from the *TopologyElementType* which is the root *Object* type of the *Device Object* type hierarchy. Objects of the type *FunctionalGroupType* are used to group the *Parameters* and Methods of the Device into logical groups. The *FunctionalGroupType* and the grouping concept are defined in UA Part DI but the groups are Device type specific i.e. the groups ProcessData and Configuration are defined by the *TemperatureControllerType* in this example.
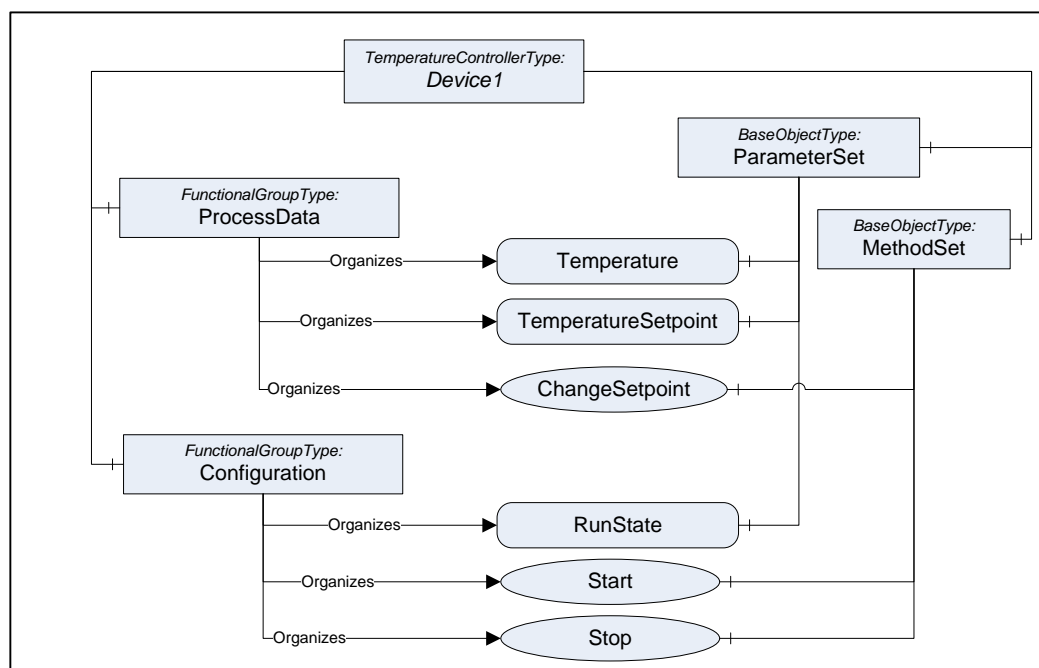
**Figure 5 – OPC UA Devices Example**

Another UA Part DI concept used in this specification is described in Figure 6. The *ConfigurableObjectType* is used to provide a way to group sub components of a Device and to indicate which types of sub components can be instantiated. The allowed types are referenced from the *SupportedTypes* folder. This information can be used by configuration clients to allow a user to select the type to instantiate as sub component of the *Device*.
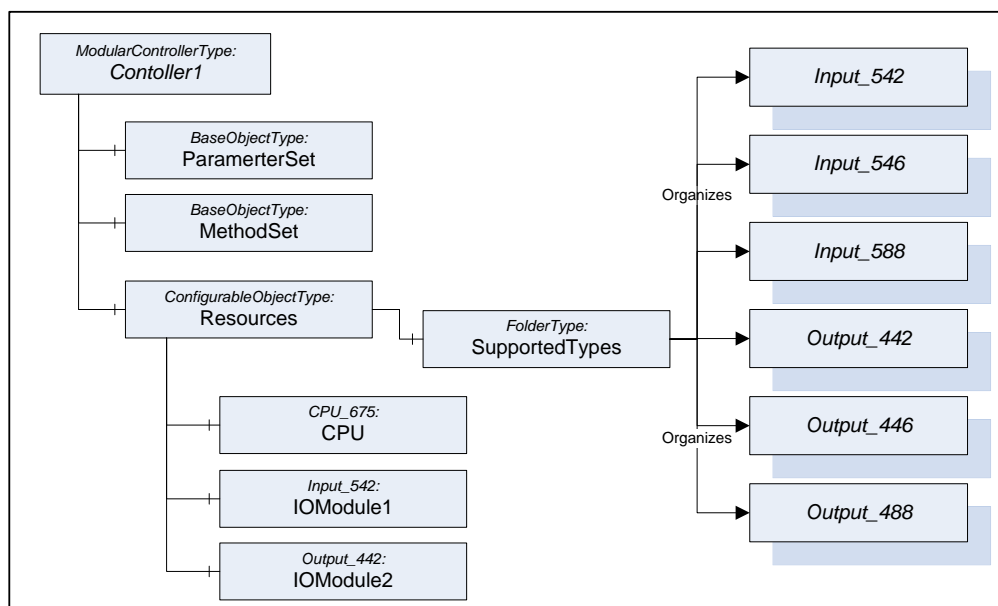


**Figure 6 – OPC UA Devices Example**

The *SupportedTypes* Folder can contain different subsets of *ObjectTypes* for different instances of the *ModularControlerType* depending on their current configuration since the list contains only types that can be instantiated for the current configuration. The example expects that only one CPU can be used in the *ModularControlerType* and this CPU is already configured. The *SupportedTypes* Folder on the *ModularControlerType* contains all possible types including CPU types that can be used in the *ModularControlerType*.

### 3.8.3  Introductory Example

A simple example shall be used to explain how the above introduced OPC UA concepts are used to represent  elements in an OPC UA Server.

According to IEC 61131-3, *Ctrl Function Blocks* consist of a name, *Ctrl Variables* with associated data types (input, output, internal), and a body containing the algorithm to be executed. These data are represented by OPC UA *ObjectTypes* derived from the *ObjectType CtrlFunctionBlockType* (see Figure 9).

To start, IEC 61131-3 requires a *Ctrl Function Block* type declaration. Here an integer up-counter is used which follows the  standard counter *Ctrl Function Block*. CTU_INT contains three input *Ctrl Variables* (CU – counter up, R – reset, PV – primary value), one local *Ctrl Variable* (PVmax) and two output *Ctrl Variables* (Q, CV – counter value) with their respective data types. Furthermore, CTU_INT has a body containing the algorithm to do the actual counting. The formal declaration of CTU_INT using the Structured Text programming language is shown in Figure 7.

```
FUNCTION_BLOCK CTU_INT

VAR_INPUT
  CU: BOOL;
  R: BOOL;
  PV: INT;
END_VAR

VAR
  PVmax: INT := 32767;
END_VAR

VAR_OUTPUT
  Q: BOOL;
  CV: INT;
END_VAR

  IF R THEN
    CV := 0;
  ELSIF CU AND (CV < PVmax) THEN
    CV := CV + 1;
  END_IF ;
  Q := (CV >= PV);

END_FUNCTION_BLOCK
```

**Figure 7 – *Ctrl Function Block* CTU_INT declaration**

The OPC UA representation of CTU_INT is shown in Figure 9. The *ObjectType* CTU_INT is a subtype of the *ObjectType CtrlFunctionBlockType*. Its components are defined by instance declaration and referenced by *HasInputVar*, *HasLocalVar*, and *HasOutputVar References*.

After declaration of CTU_INT it is instantiated twice (MyCounter, MyCounter2) and used within a *Ctrl Program* MyTestProgram shown in Figure 8. Signal and Signal 2 are counted, the *Ctrl Function Block* output *Ctrl Variables* are transferred to some temporary *Ctrl Variables* but are not further processed in this example.

```
PROGRAM MyTestProgram

VAR_INPUT
  Signal: BOOL;
  Signal2: BOOL;
END_VAR

VAR
  MyCounter: CTU_INT;
  MyCounter2: CTU_INT;
END_VAR

VAR_TEMP
  QTemp: BOOL;
  CVTemp: INT;
END_VAR

  MyCounter(CU := Signal, R := FALSE, PV := 24);

  QTemp := MyCounter.Q;
  CVTemp := MyCounter.CV;


  MyCounter2(CU := Signal2, R := FALSE, PV := 19);

  QTemp := MyCounter2.Q;
  CVTemp := MyCounter2.CV;

END_PROGRAM
```

**Figure 8 – *Ctrl Function Block* MyCounter / MyCounter2 instantiation and usage**

The OPC UA representation of the *Objects* MyCounter and MyCounter2 is shown in Figure 9. The *Objects* are instances of the *ObjectType* CTU_INT which is indicated by the *HasTypeDefinition References*. The example specific *ObjectType* CTU_INT is derived from the *ObjectType CtrlFunctionBlockType* which is indicated by the *HasSubType Reference*. Current values at a certain point in time are provided by the instances, e. g. the current counter value of MyCounter equals 11. The *Ctrl Program* MyTestProgram is not represented in the figure.
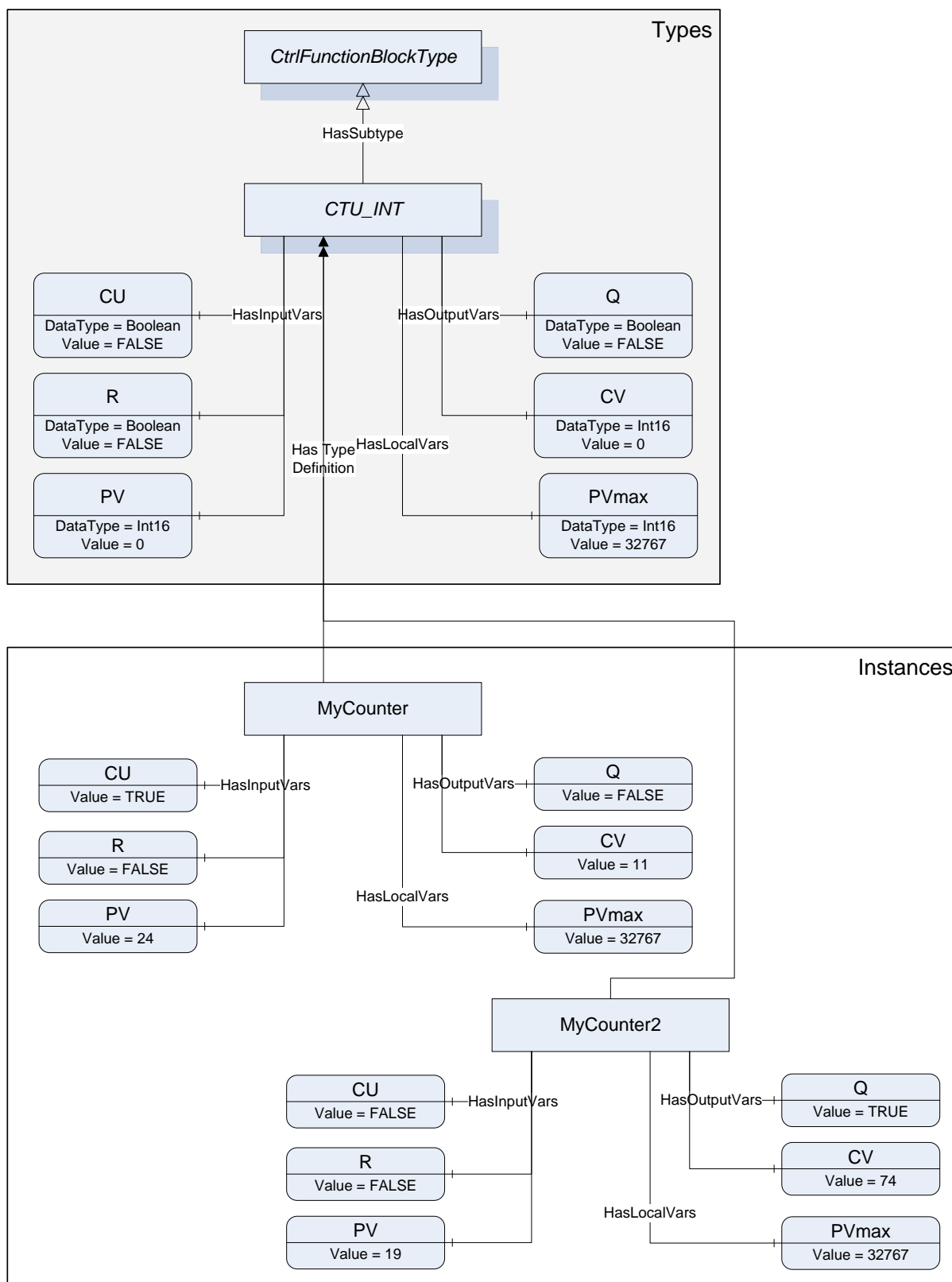
**Figure 9 – Introductory Example – OPC UA representation**

### 3.8.4  Use Cases

The following use cases illustrate the usage of the information model. Not all necessary *Objects* must be realized within a concrete OPC UA Server.

- Observation

  Observation comprises reading and monitoring data of *Ctrl Configurations, Ctrl Resources, Ctrl Tasks, Ctrl Programs, Ctrl Function Blocks, Ctrl Variables*, and their *ObjectTypes* represented in the OPC UA Server.

  Example 1: In a brewery, several tanks of the same type are operated. They are controlled by the same *Ctrl Function Block* which is instantiated in the *Controller* once for each tank. For developing the visualization it is useful to create first a template for operating a tank, which is based on the tank *CtrlFunctionBlockType* provided by the OPC UA server. Then this template can be instantiated and connected to the *Ctrl Function Block* instances within the OPC UA server as often as required (reuse).

  Example 2: In a brewery, the number of bottles produced in the current shift shall be presented on a visualization panel. The bottles are counted by the *Controller* and the result provided as an output *Ctrl Variable* of a *Ctrl Function Block*. The visualization panel subscribes to the corresponding *Variable* in the OPC UA server, gets the current number of bottles delivered each time it is changing, and presents it to the user.

- Operation

  Operation inherits the functionality of observation and extends it.

  Operation comprises writing data of *Ctrl Variables* represented in the OPC UA Server and execution control of *Ctrl Programs* and *Ctrl Function Blocks* using *Ctrl Tasks* represented in the OPC UA Server.

  Example: In a brewery, several recipes are used to produce different kinds of beer. To choose the recipe for the next batch, the number of that recipe is written from an HMI to an input *Ctrl Variable* of a *Ctrl Function Block* via a corresponding *Variable* in the OPC UA server. After this, the batch is started using a *Ctrl Task* in the OPC UA server which triggers the *Ctrl Function Block*.

- Engineering (Programming / Maintenance)

  Engineering inherits the functionality of operation and extends it.

  Engineering comprises writing of *Ctrl Configurations, Ctrl Resources, Ctrl Tasks, Ctrl Programs, Ctrl Function Blocks, Ctrl Functions, Ctrl Variables*, and their *ObjectTypes* into the OPC UA Server.

  Example: The *Ctrl Program* of a machine tool shall be updated via remote access (internet). This download is done using programming software by writing the corresponding *Ctrl Program ObjectType* into the OPC UA server while observing strict security (and safety) regulations.

- Service

  Service inherits the functionality of engineering and extends it.

  Service comprises the carrying out of service specific functions, e. g. reading / writing of special data and firmware updates.

The following Figure 10 shows the use case diagram.
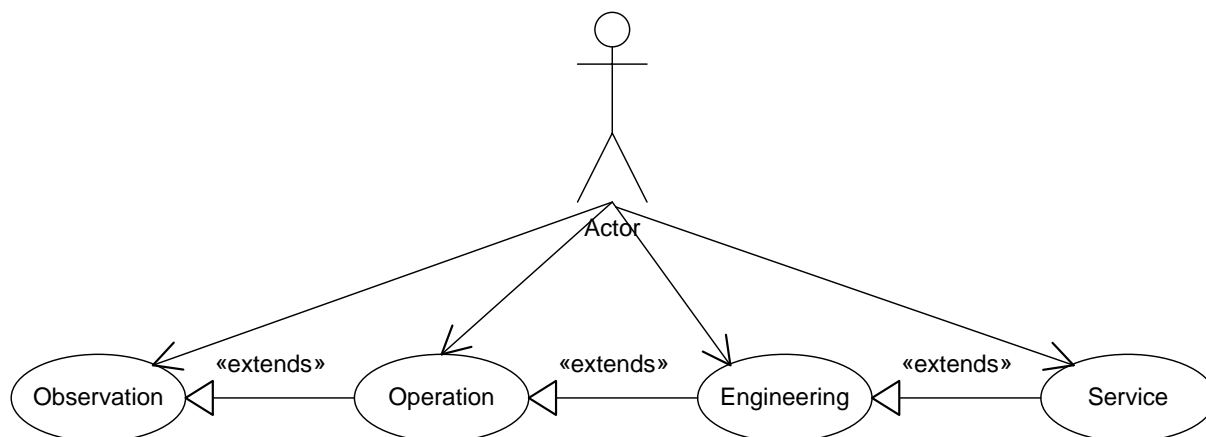
**Figure 10 – Use case diagram**

## 3.9   Conventions used in this document

### 3.9.1  Conventions for Node descriptions

*Node* definitions are specified using tables (See Table 1)

**Table 1 – Type Definition Table**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| Attribute name | Attribute value. If it is an optional Attribute that is not set "--" will be used. | | | | |
| | | | | | |
| **References** | **NodeClass** | **BrowseName** | **DataType** | **TypeDefinition** | **ModellingRule** |
| *ReferenceType* name | *NodeClass* of the Target*Node*. | *BrowseName* of the target *Node*. If the *Reference* is to be instantiated by the server, then the value of the target Node's BrowseName is "--". | *Attributes* of the referenced *Node*, only applicable for *Variables* and *Objects*. | | Referenced *ModellingRule* of the referenced *Object*. |
| Notes – Notes referencing footnotes of the table content. | | | | | |

*Attributes* are defined by providing the *Attribute* name and a value, or a description of the value.

*References* are defined by providing the *ReferenceType* name, the *BrowseName* of the *TargetNode* and its *NodeClass*.

- If the *TargetNode* is a component of the *Node* being defined in the table the *Attributes* of the composed *Node* are defined in the same row of the table. That implies that the referenced *Node* has a *HasModelParent Reference* with the *Node* defined in the Table as *TargetNode* (see UA Part 3 for the definition of *ModelParents*).

- The *DataType* is only specified for *Variables*; "[<number>]" indicates a single-dimensional array, for multi-dimensional arrays the expression is repeated for each dimension (e.g. [2][3] for a two-dimensional array). For all arrays the *ArrayDimensions* is set as identified by <number> values. If no <number> is set, the corresponding dimension is set to 0, indicating an unknown size. If no number is provided at all the *ArrayDimensions* can be omitted. If no brackets are provided, it identifies a scalar *DataType* and the *ValueRank* is set to the corresponding value (see UA Part 3). In addition, *ArrayDimensions* is set to null or is omitted. If it can be Any or

ScalarOrOneDimension, the value is put into "{<value>}", so either "{Any}" or "{ScalarOrOneDimension}" and the *ValueRank* is set to the corresponding value (see UA Part 3) and the *ArrayDimensions* is set to null or is omitted. In Table 2 examples are given.

**Table 2 – Examples of DataTypes**

| Notation | Data-Type | Value-Rank | Array-Dimensions | Description |
|---|---|---|---|---|
| Int32 | Int32 | -1 | omitted or NULL | A scalar Int32 |
| Int32[] | Int32 | 1 | omitted or {0} | Single-dimensional array of Int32 with an unknown size |
| Int32[][] | Int32 | 2 | omitted or {0,0} | Two-dimensional array of Int32 with unknown sizes for both dimensions |
| Int32[3][] | Int32 | 2 | {3,0} | Two-dimensional array of Int32 with a size of 3 for the first dimension and an unknown size for the second dimension |
| Int32[5][3] | Int32 | 2 | {5,3} | Two-dimensional array of Int32 with a size of 5 for the first dimension and a size of 3 for the second dimension |
| Int32{Any} | Int32 | -2 | omitted or NULL | An Int32 where it is unknown if it is scalar or array with any number of dimensions |
| Int32{ScalarOrOneDimension} | Int32 | -3 | omitted or NULL | An Int32 where it is either a single-dimensional array or a scalar |

- The TypeDefinition is specified for *Objects* and *Variables*.

- The TypeDefinition column specifies a *NodeId* of a *TypeDefinitionNode*, i.e. the specified *Node* points with a *HasTypeDefinition Reference* to the corresponding *TypeDefinitionNode*. The symbolic name of the *NodeId* is used in the table.

- The *ModellingRule* of the referenced component is provided by specifying the symbolic name of the rule in the *ModellingRule* column. In the *AddressSpace*, the *Node* shall use a *HasModellingRule Reference* to point to the corresponding *ModellingRule Object*.

If the *NodeId* of a *DataType* is provided, the symbolic name of the *Node* representing the *DataType* shall be used.

*Nodes* of all other *NodeClasses* cannot be defined in the same table; therefore only the used *ReferenceType*, their *NodeClass* and their *BrowseName* are specified. A reference to another of this document points to their definition.

If no components are provided, the DataType, TypeDefinition and ModellingRule columns may be omitted and only a Comment column is introduced to point to the *Node* definition.

Components of *Nodes* can be complex, i.e. containing components by themselves. The TypeDefinition, NodeClass, DataType and ModellingRule can be derived from the type definitions, and the symbolic name can be created as defined in 3.9.2.1. Therefore those containing components are not explicitly specified; they are implicitly specified by the type definitions.

### 3.9.2  NodeIds and BrowseNames

### 3.9.2.1  NodeIds

The *NodeIds* of all *Nodes* described in this document are only symbolic names. Annex A defines the actual *NodeIds*.

The symbolic name of each *Node* defined in this document is its *BrowseName*, or, when it is part of another *Node*, the *BrowseName* of the other *Node*, a ".", and the *BrowseName* of itself.

In this case "part of" means that the whole has a *HasProperty* or *HasComponent Reference* to its part. Since all *Nodes* not being part of another *Node* have a unique name in this document, the symbolic name is unique. For example, the *CtrlTaskType* defined in 4.5 has the symbolic name "TaskType". One of its *InstanceDeclarations* would be identified as "CtrlTaskType.Priority".

The namespace for this specification is defined in Annex A. The NamespaceIndex for all *NodeIds* defined in this specification is server specific and depends on the position of the namespace URI in the server namespace table.

Note: This specification does not only define concrete *Nodes*, but also requires that some Nodes have to be generated, for example one for each *Ctrl Function Block* type available in the *Controller*. The *NodeIds* of those *Nodes* are server-specific, including the Namespace. But the NamespaceIndex of those *Nodes* cannot be the NamespaceIndex used for the Nodes defined by this specification, because they are not defined by IEC 61131-3 but generated by the Server.

### 3.9.2.2  BrowseNames

The text part of the *BrowseNames* for all *Nodes* defined in this specification is specified in the tables defining the *Nodes*. The NamespaceIndex for all *BrowseNames* defined in this specification is server specific and depends on the postion of the namespace URI in the server namespace table.

### 3.9.3  Common Attributes

### 3.9.3.1  General

For all *Nodes* specified in this part, the *Attributes* named in Table 3 shall be set as specified in the table.

**Table 3 – Common Node Attributes**

| Attribute | Value |
|---|---|
| DisplayName | The *DisplayName* is a *LocalizedText*. Each server shall provide the *DisplayName* identical to the *BrowseName* of the *Node* for the LocaleId "en". Whether the server provides translated names for other LocaleIds is vendor specific. |
| Description | Optionally a vendor specific description is provided |
| NodeClass | Shall reflect the *NodeClass* of the *Node* |
| NodeId | The *NodeId* is described by *BrowseNames* as defined in 3.9.2.1 and defined in Annex A. |
| WriteMask | Optionally the *WriteMask Attribute* can be provided. If the *WriteMask Attribute* is provided, it shall set all *Attributes* to not writeable that are not said to be vendor-specific. For example, the *Description Attribute* may be set to writeable since a Server may provide a server-specific description for the *Node*. The *NodeId* shall not be writeable, because it is defined for each *Node* in this specification. |
| UserWriteMask | Optionally the *UserWriteMask Attribute* can be provided. The same rules as for the *WriteMask Attribute* apply. |

### 3.9.3.2  Objects

For all *Objects* specified in this part, the *Attributes* named in Table 4 shall be set as specified in the table.

**Table 4 – Common Object Attributes**

| Attribute | Value |
|---|---|
| EventNotifier | Whether the *Node* can be used to subscribe to *Events* or not is vendor specific |

### 3.9.3.3  Variables

For all *Variables* specified in this part, the *Attributes* named in Table 5 shall be set as specified in the table.

**Table 5 – Common Variable Attributes**

| Attribute | Value |
|---|---|
| MinimumSamplingInterval | Optionally, a vendor-specific minimum sampling interval is provided |
| AccessLevel | The access level for *Variables* used for type definitions is vendor-specific, for all other *Variables* defined in this part, the access level shall allow a current read; other settings are vendor specific. |
| UserAccessLevel | The value for the *UserAccessLevel Attribute* is vendor-specific. It is assumed that all *Variables* can be accessed by at least one user. |
| Value | For *Variables* used as *InstanceDeclarations,* the value is vendor-specific; otherwise it shall represent the value described in the text. |
| ArrayDimensions | If the *ValueRank* does not identify an array of a specific dimension (i.e. *ValueRank* <= 0) the *ArrayDimensions* can either be set to null or the *Attribute* is missing. This behaviour is vendor-specific.<br>If the *ValueRank* specifies an array of a specific dimension (i.e. *ValueRank* > 0) then the *ArrayDimensions Attribute* shall be specified in the table defining the *Variable*. |

### 3.9.3.4  VariableTypes

For all *VariableTypes* specified in this part, the *Attributes* named in Table 6 shall be set as specified in the table.

**Table 6 – Common VariableType Attributes**

| Attributes | Value |
|---|---|
| Value | Optionally a vendor-specific default value can be provided |
| ArrayDimensions | If the *ValueRank* does not identify an array of a specific dimension (i.e. *ValueRank* <= 0) the *ArrayDimensions* can either be set to null or the *Attribute* is missing. This behaviour is vendor-specific.<br>If the *ValueRank* specifies an array of a specific dimension (i.e. *ValueRank* > 0) then the *ArrayDimensions Attribute* shall be specified in the table defining the *VariableType*. |

### 3.9.4  Reference to IEC 61131-3 Definitions

Referenced key words in this document like VAR_GLOBAL are defined in the IEC 61131-3 specification. See table delimiters and key words in the IEC 61131-3 for a complete list.

## 4   Model

### 4.1   General

Figure 11 depicts the main *ObjectTypes* of this specification and their relationships. The drawing is not intended to be complete. For the sake of simplicity only a few components and relations were captured so as to give a rough idea of the overall structure of the *IEC 61131-3 Information Model*.
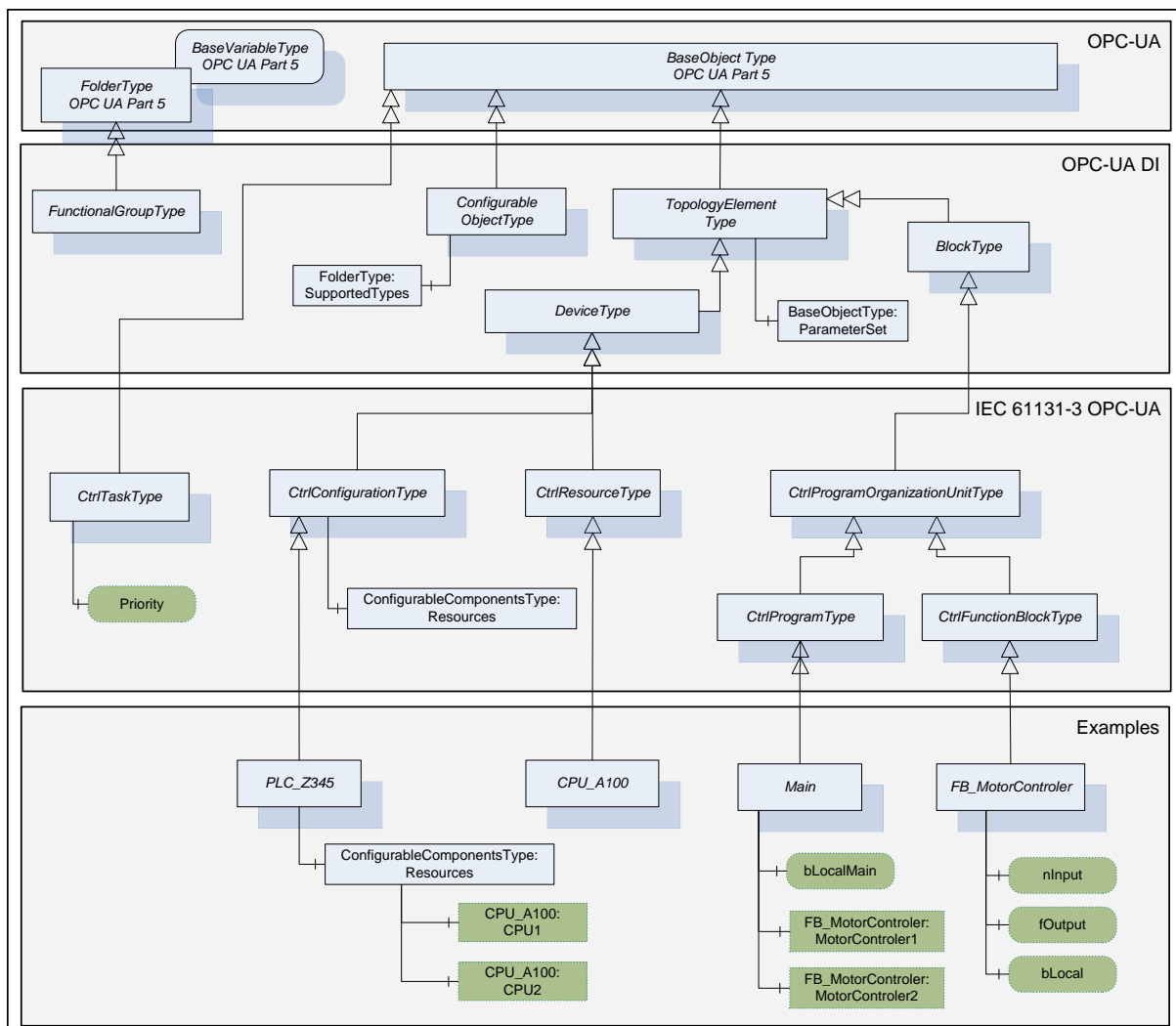


**Figure 11 – *OPC UA IEC 61131-3 ObjectTypes* Overview**

The boxes in this drawing show the *ObjectTypes* used in this specification as well as some elements from other specifications that help understand the overall context. The upper grey box shows the OPC UA core *ObjectTypes* from which the OPC UA Device Integration Types are derived. The Device Integration model and its Types in the second level are used as base for the *IEC 61131-3 ObjectTypes*. The grey box in the third level shows the *IEC 61131-3 ObjectTypes* that this specification introduces. The components of those *ObjectTypes* are illustrated only in an abstract way in this overall picture. The grey box in the lowest level represents examples of sub types defined by vendors or *Controller* programmers.

Typically, the components of an *ObjectType* are fixed and can be extended by subtyping. However, since each *Object* of an *ObjectType* can be extended with additional components, this specification allows extending the standard *ObjectTypes* defined in this document with additional components. Thereby, it is possible to express the additional information in the type definition that would already be contained in each *Object*. Some *ObjectTypes* already provide

entry points for server specific extensions. However, it is not allowed to restrict the components of the standard *ObjectTypes* defined in this specification. An example of extending the *ObjectTypes* is putting the standard *Property NodeVersion* defined in UA Part 3 into the *BaseObjectType*, stating that each *Object* of the server will provide a *NodeVersion*.

It is not the objective to map all IEC 61131-3 constraints to the OPC UA Information Model, but to define an OPC UA Information Model which is capable to hold at least all possible data of one or more IEC 61131-3 compliant *Ctrl Configurations*.

A *Ctrl Configuration* compliant to IEC 61131-3 represents the special case of a complete engineered *Controller* with an OPC UA server providing access to all data of one or more IEC 61131-3 compliant *Ctrl Configurations.* In general, an OPC UA server may provide incomplete *Ctrl Configurations*, e.g. during the engineering process or because not all data shall be accessed from outside.

Examples for *Object* and *Variable* instances of the vendor or controller programmer specific types are shown in Figure 12. The *Root* and the *Objects Folder* are *Nodes* defined by UA Part 5. The *Objects Folder* is the main entry point for *Object* instances.
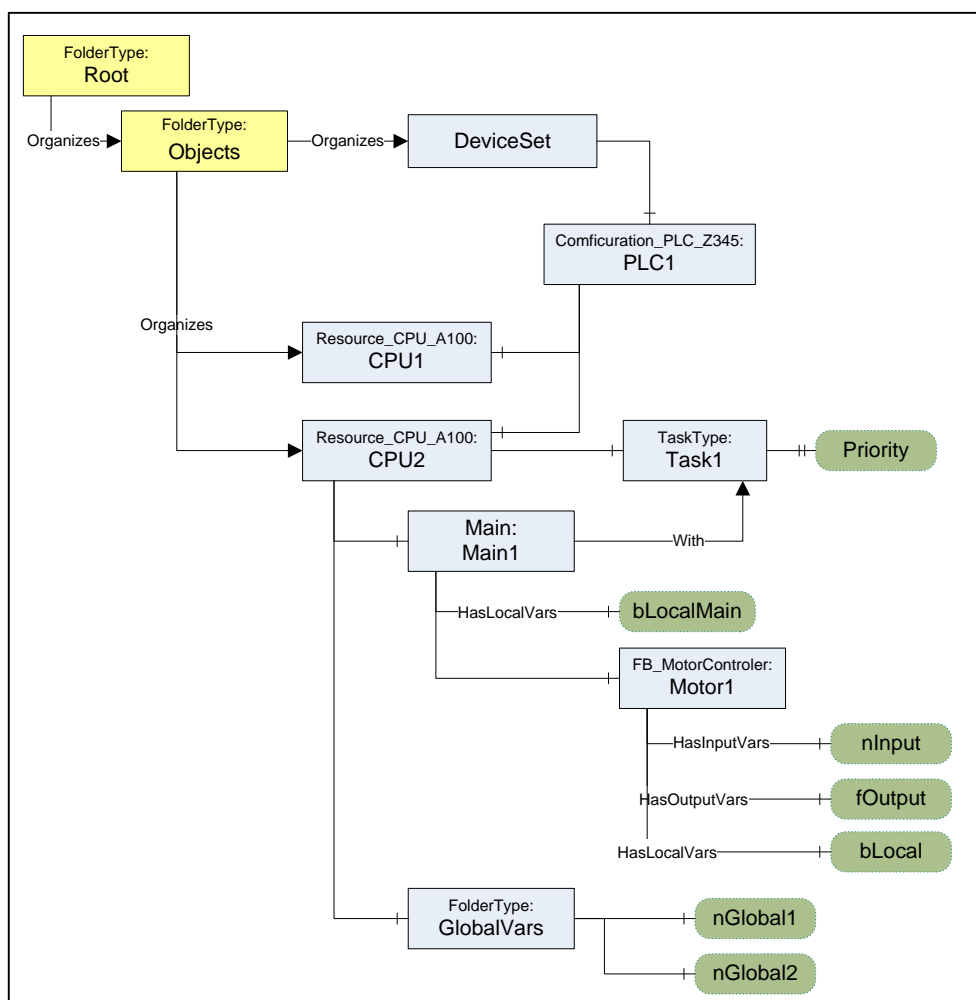
**Figure 12 – *OPC UA IEC 61131-3 Object Instance Example***

**4.2    CtrlConfigurationType**

**4.2.1  ObjectType definition**

This *ObjectType* defines the representation of a *Ctrl Configuration* of a programmable *Controller* system in an OPC UA *Address Space*. It introduces *Objects* to group *Ctrl Resources* and different types of *Ctrl Variables*. The *CtrlConfigurationType* is derived from the *TopologyElementType* defined in UA Part DI. Figure 13 shows the *CtrlConfigurationType*. It is formally defined in Table 7. The dark grey nodes in the figure are examples and are not part of the *ObjectType* definition.
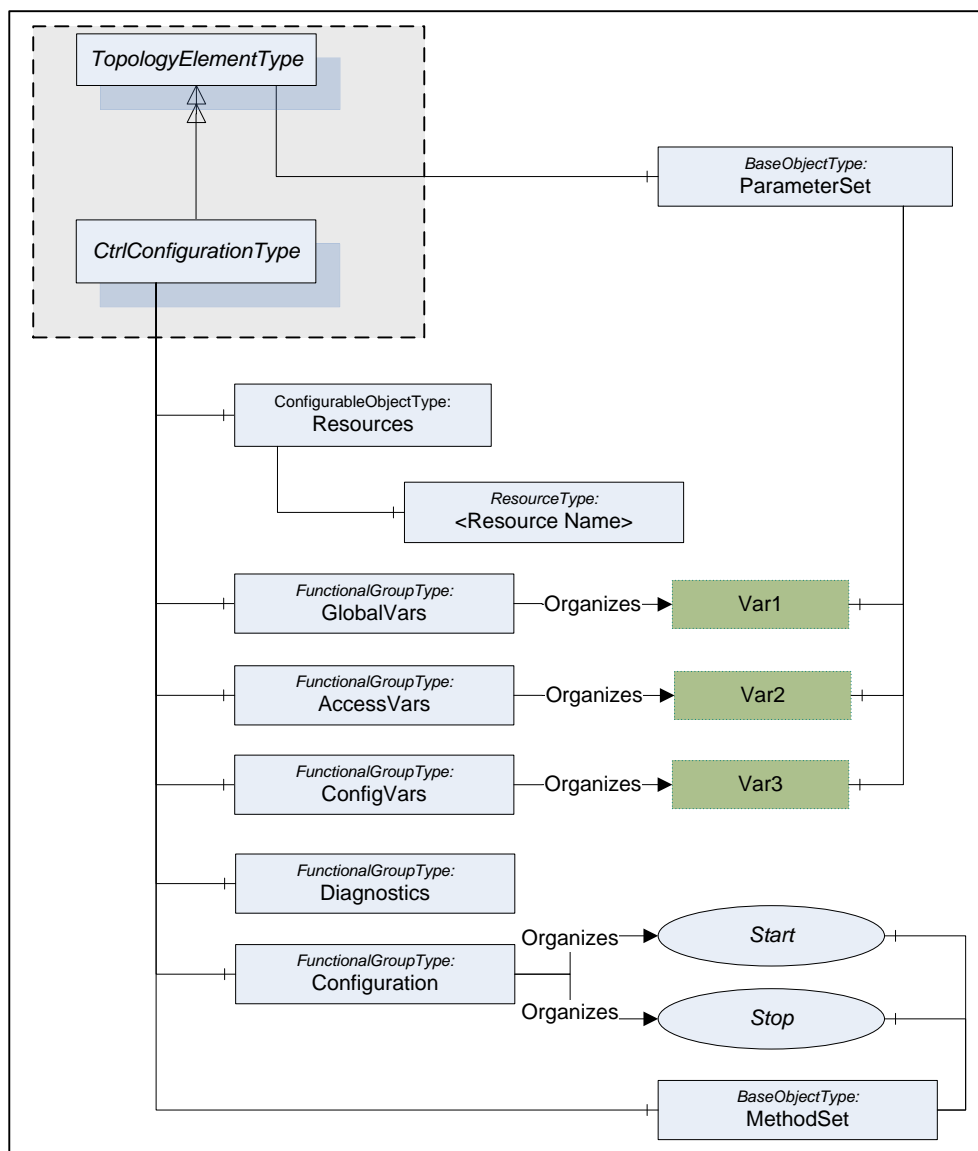
**Figure 13 – *CtrlConfigurationType* Overview**

The *CtrlConfiguration ObjectType* is formally defined in Table 7.

**Table 7 – *CtrlConfigurationType* Definition**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | CtrlConfigurationType | | | | |
| IsAbstract | False | | | | |
| **References** | **NodeClass** | **BrowseName** | **DataType** | **TypeDefinition** | **ModellingRule** |
| Inherit the components of the *TopologyElementType* defined in [UA Part DI] | | | | | |
| | | | | | |
| HasComponent | Object | MethodSet | | BaseObjectType | Optional |
| HasComponent | Object | Resources | | ConfigurableObjectType | Mandatory |
| HasComponent | Object | GlobalVars | | FunctionalGroupType | Optional |
| HasComponent | Object | AccessVars | | FunctionalGroupType | Optional |
| HasComponent | Object | ConfigVars | | FunctionalGroupType | Optional |
| HasComponent | Object | Configuration | | FunctionalGroupType | Optional |
| HasComponent | Object | Status | | FunctionalGroupType | Optional |

The *CtrlConfigurationType ObjectType* is a concrete type and can be used directly. It is recommended to create subtypes for vendor or user specific configurations.

*A concrete Ctrl Configuration* type or instance may have *ParameterSet*, *Parameters* and *FunctionalGroup*s as defined for the *TopologyElementType* in UA Part DI.

The *MethodSet Object* is defined by the *TopologyElementType* and is overwritten in the *CtrlConfigurationType* to add the *HasComponent References* to the *Methods* defined for the *CtrlConfigurationType*.

The *Object Resources* is used to group *Ctrl Resources* that are part of the *Ctrl Configuration*. It uses the concept of configurable *Objects* defined UA Part DI. It contains *Objects* of the type *CtrlResourceType* representing a *Ctrl Resource* and a Folder with possible *Ctrl Resource* types that can be instantiated in the *Ctrl Configuration*. For a complete configuration at least one resource is necessary from an IEC 61131-3 point of view but not necessary from an OPC UA point of view. Temporary, incomplete configurations are allowed, e.g. during a configuration process.

The *FunctionalGroup GlobalVars* contains the corresponding list of *Ctrl Variables* declared by the key word VAR_GLOBAL.

The *FunctionalGroup AccessVars* contains the corresponding list of *Ctrl Variables* declared by the key word VAR_ACCESS.

The *FunctionalGroup ConfigVars* contains the corresponding list of *Ctrl Variables* declared by the key word VAR_CONFIG.

The *FunctionalGroup Configuration* contains configuration *Variables* and *Methods* like start and stop.

The *FunctionalGroup Status* contains diagnostic and status information like system variables, status variables or diagnostic codes.

Starting a *Ctrl Configuration* causes the initialization of global *Ctrl Variables* and the start of all *Ctrl Resources*. Stopping a *Ctrl Configuration* stops all *Ctrl Resources*.

### 4.2.2  Resources components

The configurable *Object Resources* of the *CtrlConfigurationType* is formally defined in Table 8.

**Table 8 – Components of the *Resources Object***

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | Resources | | | | |
| **References** | **Cardinality** | **NodeClass** | **BrowseName** | **TypeDefinition** | **ModellingRule** |
| HasComponent | 0 – N | Object | <Resource Name> | *CtrlResourceType* | Optional |

### 4.2.3  MethodSet components

The *Methods* available as parts of the *CtrlConfigurationType* are formally defined in Table 9.

**Table 9 – Components of the *CtrlConfigurationType MethodSet***

| Attribute | Value | | | |
|---|---|---|---|---|
| BrowseName | MethodSet | | | |
| **References** | **NodeClass** | **BrowseName** | **Description** | **ModellingRule** |
| *Configuration FunctionalGroup* | | | | |
| The following components are also referenced from the *FunctionalGroup Configuration* using *Organizes References*. | | | | |
| HasComponent | Method | Start | This Method is used to start a *Ctrl Configuration*. Only the browse name is defined for this *Method*. The *Method* parameters are vendor specific. | Optional |
| HasComponent | Method | Stop | This Method is used to stop a *Ctrl Configuration*. Only the browse name is defined for this *Method*. The *Method* parameters are vendor specific. | Optional |

### 4.3    CtrlResourceType

### 4.3.1  ObjectType definition

This *ObjectType* defines the representation of a *Crtl Resources* of a programmable *Controller* system in an OPC UA *Address Space*. It introduces *Objects* to group Configuration and Diagnostic capabilities, *GlobalVars* and *Ctrl Programs* executed under the control of *Tasks*. The *CtrlResourceType* is derived from the *DeviceType* defined in UA Part DI. Figure 14 shows the *CtrlResourceType*. It is formally defined in Table 10. The dark grey node in the figure is an example and is not part of the *ObjectType* definition.
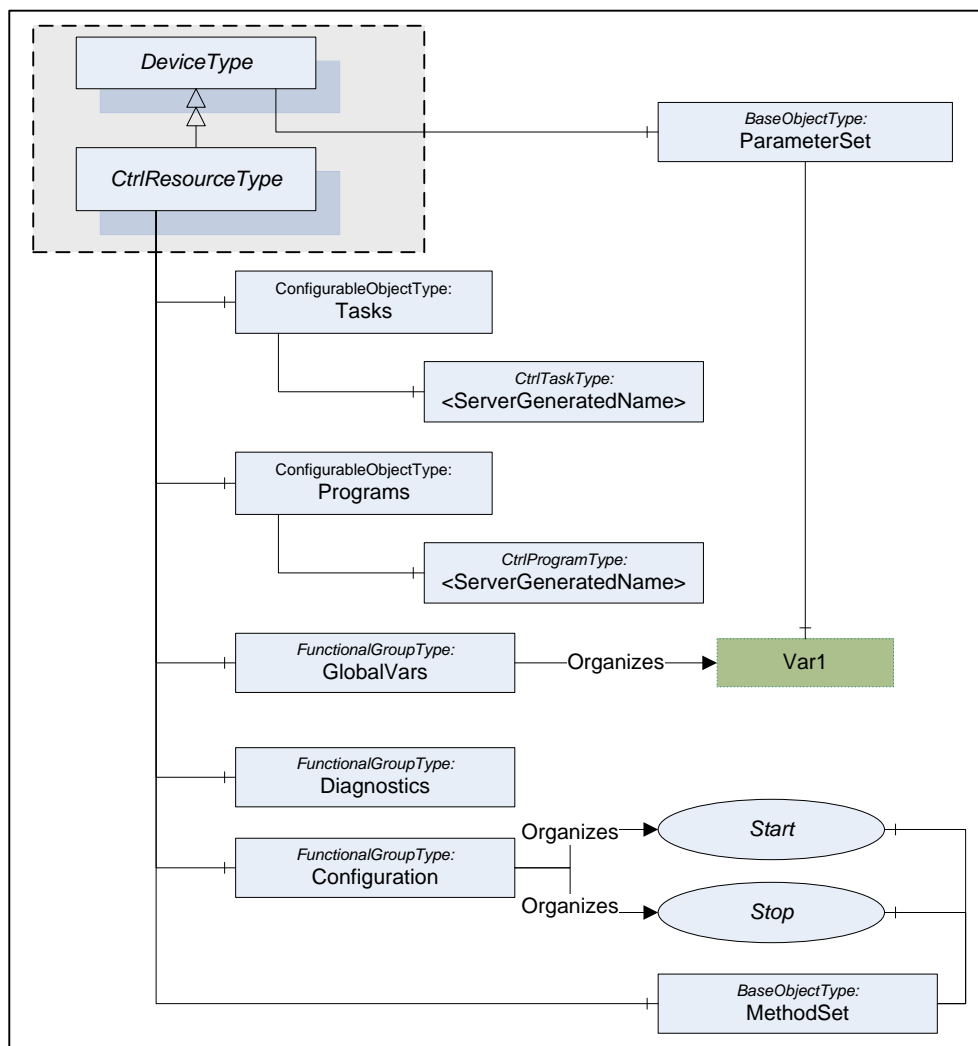
**Figure 14 – *CtrlResourceType* Overview**

The *Ctrl Resource ObjectType* is formally defined in Table 10.

**Table 10 – *CtrlResourceType* Definition**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | CtrlResourceType | | | | |
| IsAbstract | False | | | | |
| **References** | **NodeClass** | **BrowseName** | **DataType** | **TypeDefinition** | **ModellingRule** |
| Inherit the components of the *DeviceType* defined in [UA Part DI] | | | | | |
| | | | | | |
| HasComponent | Object | MethodSet | | BaseObjectType | Optional |
| HasComponent | Object | Tasks | | ConfigurableObjectType | Mandatory |
| HasComponent | Object | Programs | | ConfigurableObjectType | Mandatory |
| HasComponent | Object | GlobalVars | | FunctionalGroupType | Optional |
| HasComponent | Object | Configuration | | FunctionalGroupType | Optional |
| HasComponent | Object | Status | | FunctionalGroupType | Optional |

The *CtrlResourceType ObjectType* is a concrete type and can be used directly. It is recommended to create subtypes for vendor or user specific resources.

*A concrete Ctrl Resource* type or instance may have *ParameterSet*, *Parameters* and *FunctionalGroup*s as defined for the *TopologyElementType* in UA Part DI.

The *MethodSet Object* is defined by the *TopologyElementType* and is overwritten in the *CtrlResourceType* to add the *HasComponent References* to the *Methods* defined for the *CtrlResourceType*.

The *Object Tasks* is used to group *Ctrl Tasks* that are part of the *Ctrl Resource*. It uses the concept of configurable *Objects* defined UA Part DI. It contains *Objects* of the type *CtrlTaskType* representing a *Ctrl Resource* and a Folder with possible *Ctrl Task* types that can be instantiated in the *Ctrl Resource*.

The configurable *Object Programms* is used to group *Ctrl Programs* that are part of the *Ctrl Resource*. It contains *Objects* of the type *CtrlTaskType* representing a *Ctrl Resource* and a Folder with possible *Ctrl Program* types that can be instantiated in the *Ctrl Resource*.

The *FunctionalGroup GlobalVars* contains the corresponding list of *GlobalVars* that may be accessed in the programmable *Controller* system within the scope of the *Ctrl Resource*.

The *FunctionalGroup Configuration* contains configuration *Variables* and *Methods* like start and stop.

The *FunctionalGroup Status* contains diagnostic and status information like system variables, system events or diagnostic codes.

### 4.3.2  Tasks components

The configurable *Object Tasks* of the *CtrlResourceType* is formally defined in Table 11.

**Table 11 – Components of the *Tasks Object***

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | Tasks | | | | |
| **References** | **Cardinality** | **NodeClass** | **BrowseName** | **TypeDefinition** | **ModellingRule** |
| HasComponent | 0 – N | Object | &lt;Task Name&gt; | *CtrlTaskType* | Optional |

### 4.3.3  Programs components

The configurable *Object Programs* of the *CtrlResourceType* is formally defined in Table 12.

**Table 12 – Components of the *Programs Object***

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | Programs | | | | |
| **References** | **Cardinality** | **NodeClass** | **BrowseName** | **TypeDefinition** | **ModellingRule** |
| HasComponent | 0 – N | Object | &lt;Program Name&gt; | *CtrlProgramType* | Optional |

### 4.3.4  MethodSet components

The *Methods* available as parts of the *CtrlResourceType* are formally defined in Table 13.

**Table 13 – Components of the *CtrlResourceType MethodSet***

| Attribute | Value | | | |
|---|---|---|---|---|
| BrowseName | MethodSet | | | |
| **References** | **NodeClass** | **BrowseName** | **Description** | **ModellingRule** |
| *Configuration FunctionalGroup* | | | | |
| The following components are also referenced from the *FunctionalGroup Configuration* using *Organizes References*. | | | | |
| HasComponent | Method | Start | This Method is used to start a *Ctrl Resource*. Only the browse name is defined for this *Method*. The *Method* parameters are vendor specific. | Optional |
| HasComponent | Method | Stop | This Method is used to stop a *Ctrl Resource*. Only the browse name is defined for this *Method*. The *Method* parameters are vendor specific. | Optional |

## 4.4   CtrlProgramOrganizationUnitType

This *ObjectType* defines the representation of a *Ctrl Program Organization Unit* of a programmable *Controller* system in an OPC UA *Address Space*. It defines how components of the *Ctrl Program Organization Unit like Variables* and *Ctrl Function Blocks* are represented. The *CtrlProgramOrganizationUnitType* is derived from the *BlockType* defined in UA Part DI. Figure 15 shows the *CtrlProgramOrganizationUnitType*. It is formally defined in Table 14.
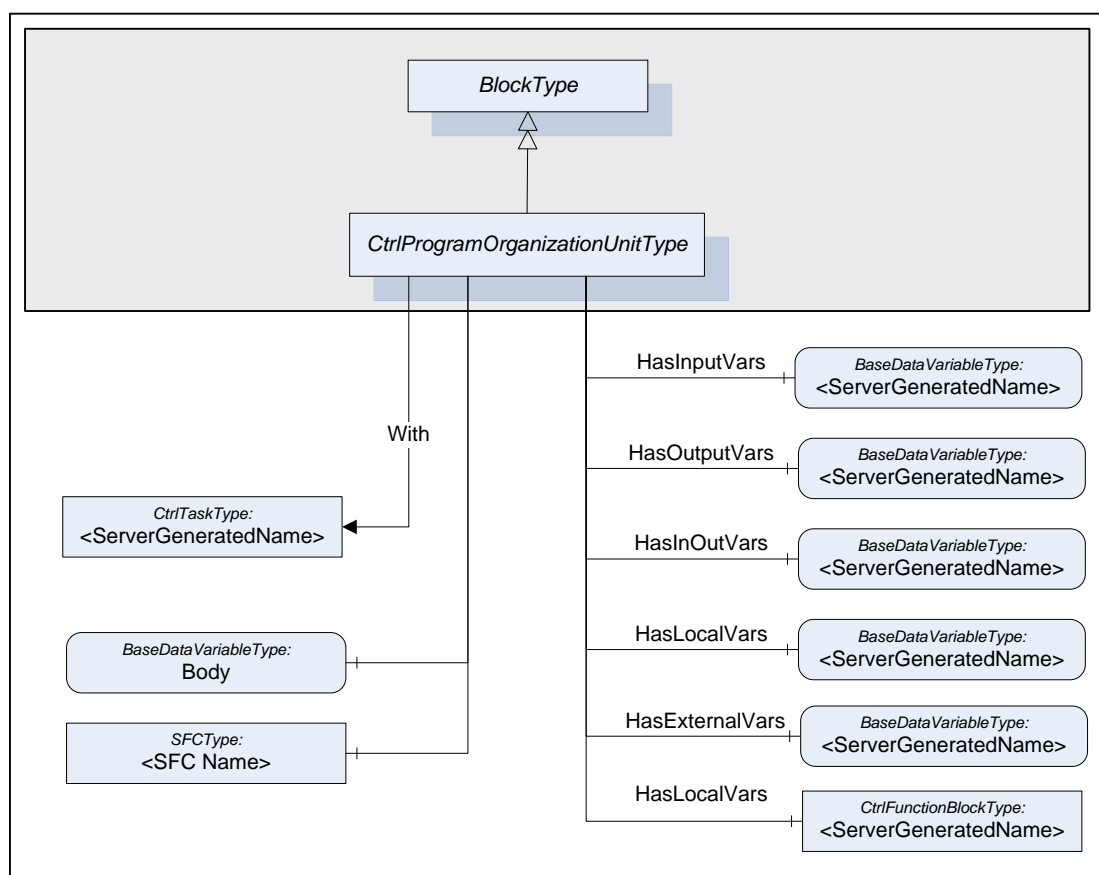


**Figure 15 – *CtrlProgramOrganizationUnitType* Overview**

The *Ctrl Program Organization Unit ObjectType* is formally defined in Table 14.

**Table 14 – *CtrlProgramOrganizationUnitType* Definition**

| Attribute | Value | | | | | |
|---|---|---|---|---|---|---|
| BrowseName | CtrlProgramOrganizationUnitType | | | | | |
| IsAbstract | True | | | | | |
| References | Cardinality | NodeClass | BrowseName | DataType | TypeDefinition | Modelling Rule |
| Inherit the components of the *BlockType* defined in [UA Part DI] | | | | | | |
| HasSubtype | | ObjectType | CtrProgramType | Defined in Clause 4.4.1 | | |
| HasSubtype | | ObjectType | CtrlFunctionBlock Type | Defined in Clause 4.4.2 | | |
| | | | | | | |
| With | 0 – N | Object | <Taks Name> | | CtrlTaskType | Optional |
| HasInputVar | 0 – N | Variable | <Var Name> | BaseDataType | BaseDataVariableType | Optional |
| HasOutputVar | 0 – N | Variable | <Var Name> | BaseDataType | BaseDataVariableType | Optional |
| HasInOutVar | 0 – N | Variable | <Var Name> | BaseDataType | BaseDataVariableType | Optional |
| HasLocalVar | 0 – N | Variable | <Var Name> | BaseDataType | BaseDataVariableType | Optional |
| HasExternalVar | 0 – N | Variable | <Var Name> | BaseDataType | BaseDataVariableType | Optional |
| HasLocalVar | 0 – N | Object | <Block Name> | | CtrlFunctionBlockType | Optional |
| HasComponent | | Variable | Body | XmlElement | BaseDataVariableType | Optional |
| HasComponent | 0 – N | Object | <SFC Name> | | SFCType | Optional |

The *CtrlProgramOrganizationUnitType ObjectType* is abstract. It is the common base type for all *Ctrl Program Organization Unit* specific types.

The *With Reference* defined in 4.7.7 is used to reference the Ctrl Task that is used to execute the *Ctrl Program Organization Unit*.

*Variables* declared for a *Ctrl Program Organization Unit* type are referenced with different subtypes of the *HasComponent Reference*. The used *Reference* type depends on the IEC 61131-3 variable declaration keywords. The characteristics of the Variables like data type and access rights and their mapping from IEC 61131-3 information and key words is defined in chapter 5. The name of the Variable depends on the *Variable* name in the *Ctrl Program Organization Unit*.

Variables declared with the key word VAR_INPUT are referenced with *HasInputVar* defined in 4.7.2.

Variables declared with the key word VAR_OUTPUT are referenced with *HasOutputVar* defined in 4.7.3.

Variables declared with the key word VAR_IN_OUT are referenced with *HasInOutVar* defined in 4.7.4.

Variables declared with the key word VAR are referenced with *HasLocalVar* defined in 4.7.5.

Variables declared with the key word VAR_EXTERNAL are referenced with *HasExternalVar* defined in 4.7.6.

*Ctrl Function Blocks* declared with the key word VAR are referenced with *HasLocalVar* defined in 4.7.5. The name of the *Object* depends on the name of the block in the *Ctrl Program Organization Unit*.

The *Variable Body* contains the body of the *Ctrl Program Organisation Unit* as *XmlElement*.

Sequential function charts (SFC) declared in the *Ctrl Program Organisation Unit* are represented as Objects of the type *SFCType* defined in chapter 4.6.

### 4.4.1  CtrlProgramType

This *ObjectType* defines the representation of a *Ctrl Program* of a programmable *Controller* system in an OPC UA *Address Space*. It is derived from *CtrlProgramOrganizationUnitType* and introduces additional Variables in addition to the components of the base type. Figure 16 shows the *CtrlProgramType*. It is formally defined in Table 15.
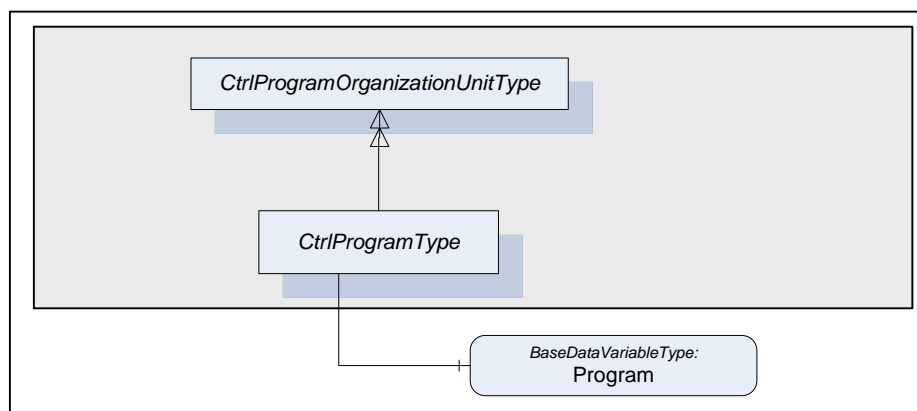
**Figure 16 – *CtrlProgramType* Overview**

The *Ctrl Program ObjectType* is formally defined in Table 15.

**Table 15 – *CtrlProgramType* Definition**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | CtrlProgramType | | | | |
| IsAbstract | True | | | | |
| **References** | **NodeClass** | **BrowseName** | **DataType** | **TypeDefinition** | **ModellingRule** |
| Inherit the *Properties* and components of the CtrlProgramOrganizationUnitType | | | | | |
| | | | | | |
| HasComponent | Variable | Program | Structure | BaseDataVariableType | Optional |

The *CtrlProgramType ObjectType* is abstract. There will be no instances of a *CtrlProgramType* itself, but there will be instances of subtypes of this type like instances of vendor or user specific *Ctrl Programs*.

The *Program Variable* component contains the complete *Ctrl Program* data in a complex *Variable*.

### 4.4.2  CtrlFunctionBlockType

This *ObjectType* defines the representation of a *Ctrl Function Blocks* of a programmable *Controller* system in an OPC UA *Address Space*. It is derived from *CtrlProgramOrganizationUnitType* and introduces *Ctrl Function Block* specific components in addition to the components of the base type. Figure 17 shows the *CtrlFunctionBlockType*. It is formally defined in Table 16.
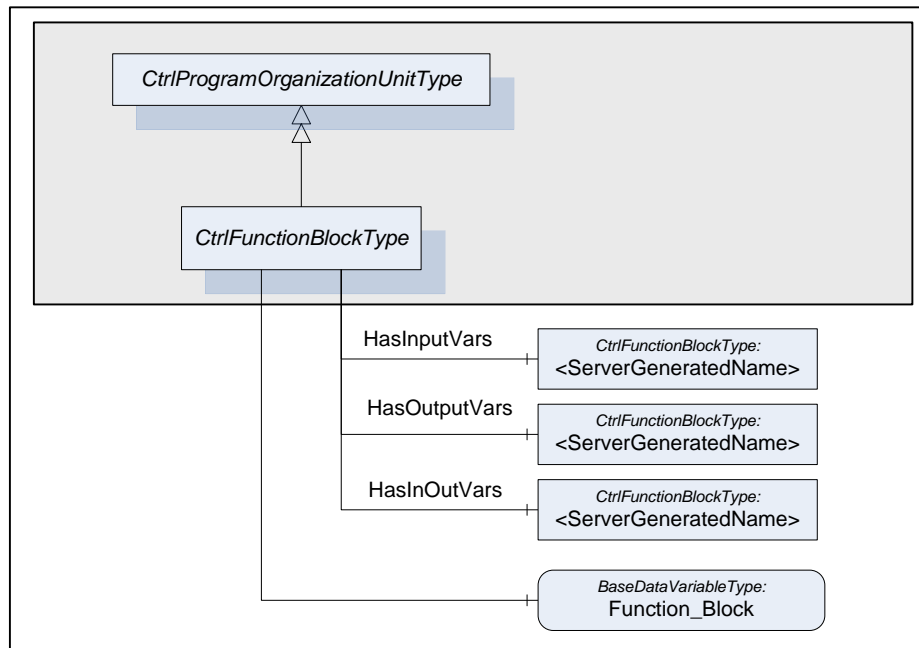


**Figure 17 – *CtrlFunctionBlockType* Overview**

The *CtrlFunctionBlock ObjectType* is formally defined in Table 16.

**Table 16 – *CtrlFunctionBlockType* Definition**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | CtrlFunctionBlockType | | | | |
| IsAbstract | True | | | | |
| **References** | **Cardinality** | **NodeClass** | **BrowseName** | **TypeDefinition** | **ModellingRule** |
| Inherit the *Properties* and components of the CtrlProgramOrganizationUnitType | | | | | |
| | | | | | |
| HasInputVar | 0 – N | Object | <server generated> | CtrlFunctionBlockType | Optional |
| HasOutputVar | 0 – N | Object | <server generated> | CtrlFunctionBlockType | Optional |
| HasInOutVar | 0 – N | Object | <server generated> | CtrlFunctionBlockType | Optional |
| HasComponent | | Variable | FunctionBlock | BaseDataVariableType | Optional |

The *CtrlFunctionBlockType ObjectType* is abstract. There will be no instances of a *CtrlFunctionBlockType* itself, but there will be instances of subtypes of this type like instances of vendor or user specific *Ctrl Function Blocks*.

*Ctrl Function Block* instances declared for a *Ctrl Function Block* type are referenced with different subtypes of the *HasComponent Reference*. The used *Reference* type depends on the IEC 61131-3 declaration keywords. The name of the *Object* depends on the name of the block in the *Ctrl Function Block*.

*Ctrl Function Block* instances declared with the key word VAR_INPUT are referenced with *HasInputVar* defined in 4.7.2.

*Ctrl Function Block* instances declared with the key word VAR_OUTPUT are referenced with *HasOutputVar* defined in 4.7.3.

*Ctrl Function Block* instances declared with the key word VAR_IN_OUT are referenced with *HasInOutVar* defined in 4.7.4.

The *FunctionBlock Variable* component contains the complete *Ctrl Function Block* data in a complex *Variable*. The *DisplayName* for the Variable is Function_Block.

## 4.5   CtrlTaskType

This *ObjectType* defines the representation of a *Ctrl Task* of a programmable *Controller* system in an OPC UA *Address Space*. It introduces Properties providing information about the *Ctrl Task*. Figure 18 shows the *CtrlTaskType*. It is formally defined in Table 17.
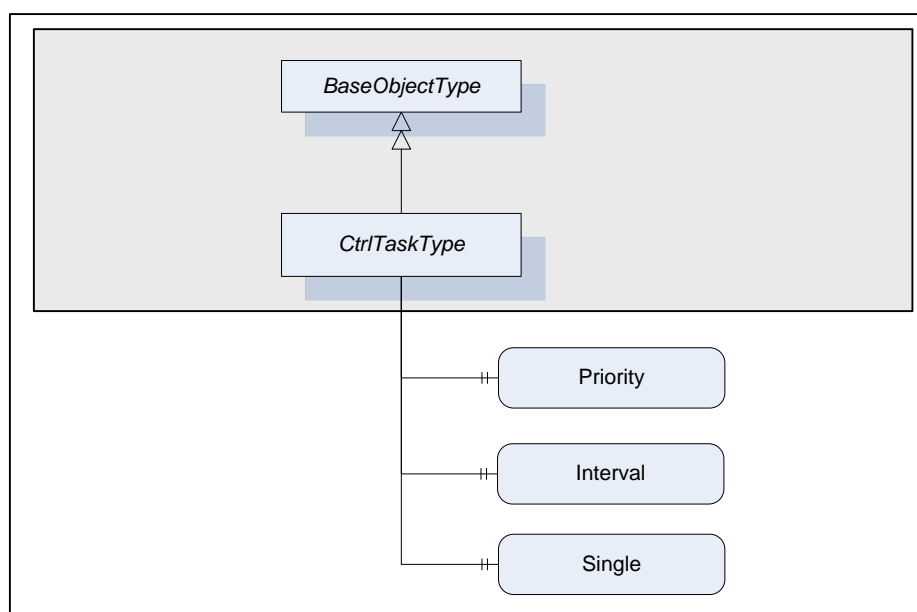


**Figure 18 – *CtrlTaskType* Overview**

The *Ctrl Task ObjectType* is formally defined in Table 17.

**Table 17 – *CtrlTaskType* Definition**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | CtrlTaskType | | | | |
| IsAbstract | False | | | | |
| **References** | **NodeClass** | **BrowseName** | **DataType** | **TypeDefinition** | **ModellingRule** |
| Inherit the *Properties* of the *BaseObjectType* defined in [UA Part 5] | | | | | |
| | | | | | |
| HasProperty | Variable | Priority | UInt32 | PropertyType | Mandatory |
| HasProperty | Variable | Interval | String | PropertyType | Optional |
| HasProperty | Variable | Single | String | PropertyType | Optional |

The *Priority Property* indicates the scheduling priority of the associated *Ctrl Program Organization Unit*.

The *Interval Property* indicates the periodical scheduling of the associated *Ctrl Program Organization Unit* at the specified interval*.*

The *Single Property* indicates a single scheduling of the associated *Ctrl Program Organization Unit* at each rising edge.

## 4.6    SFCType

The SFC *ObjectType* is formally defined in Table 18. This type is a container for Sequential Function Chart (SFC) related information. The representation of this information is vendor specific. Future versions of this specification may define standard representations.

**Table 18 – SFC*Type* Definition**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | SFCType | | | | |
| IsAbstract | False | | | | |
| **References** | **NodeClass** | **BrowseName** | **DataType** | **TypeDefinition** | **ModellingRule** |
| Inherit the *Properties* of the *BaseObjectType* defined in [UA Part 5] | | | | | |
| | | | | | |

## 4.7    Reference Types

### 4.7.1    General

Figure 19 depicts the main *ObjectTypes* of this specification and their relationship.
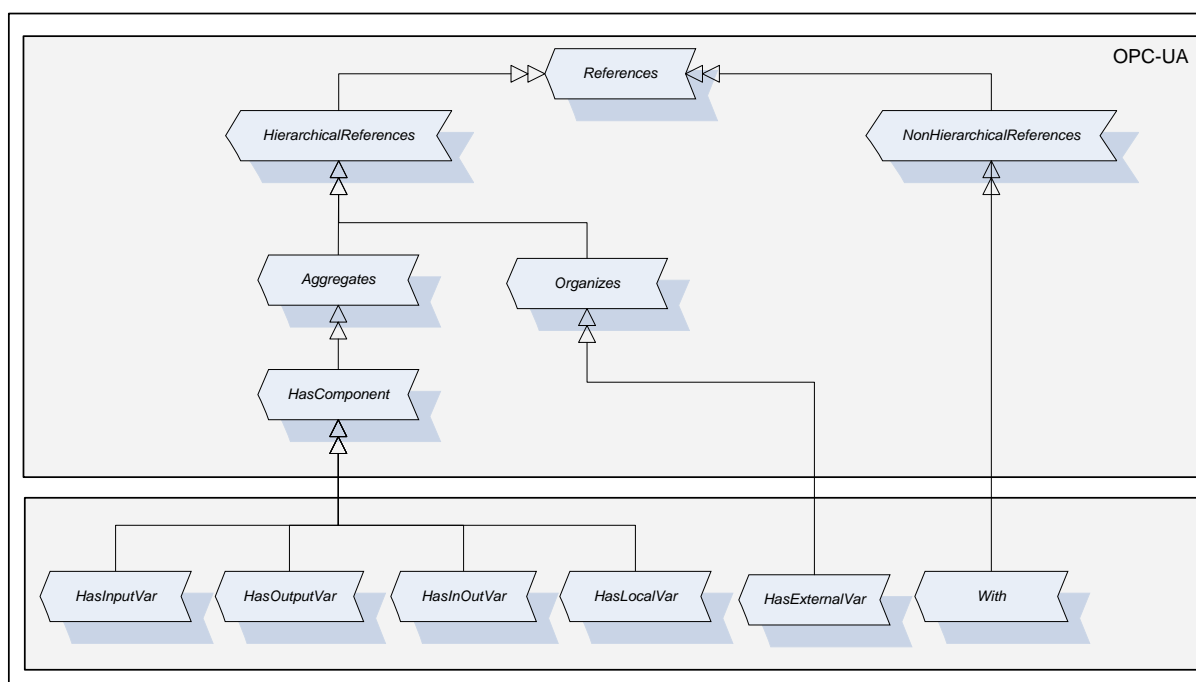


**Figure 19 – Reference Types Overview**

The upper grey box shows the OPC UA core *ReferenceTypes* from which the *IEC 61131-3 ReferenceTypes* are derived. The grey box in the second level shows the *IEC 61131-3 ReferenceTypes* that this specification introduces.

### 4.7.2    HasInputVar

This *ReferenceType* is a subtype of the *HasComponent ReferenceType* defined in UA Part 5. Its representation in the *AddressSpace* is specified in Table 19.

**Table 19 – HasInputVar ReferenceType**

| Attributes | Value | | |
|---|---|---|---|
| BrowseName | HasInputVar | | |
| InverseName | InputVarOf | | |
| Symmetric | False | | |
| IsAbstract | False | | |
| References | NodeClass | BrowseName | Comment |
| Subtype of HasComponent Reference*Type* defined in [UA Part 5] | | | |

The *HasInputVar ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference components of a *Ctrl Program Organization Unit* declared with the key word VAR_INPUT.

The *SourceNode* of *References* of this type shall be a subtype of *CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* or an *Object* of the *ObjectType CtrlFunctionBlockType* or one of its subtypes.

### 4.7.3    HasOutputVar

This *ReferenceType* is a subtype of the *HasComponent ReferenceType* defined in UA Part 5. Its representation in the *AddressSpace* is specified in Table 20.

**Table 20 – HasOutputVar ReferenceType**

| Attributes | Value | | |
|---|---|---|---|
| BrowseName | HasOutputVar | | |
| InverseName | OutputVarOf | | |
| Symmetric | False | | |
| IsAbstract | False | | |
| References | NodeClass | BrowseName | Comment |
| Subtype of HasComponent Reference*Type* defined in [UA Part 5] | | | |

The *HasOutputVar ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference components of a *Ctrl Program Organization Unit* declared with the key word VAR_OUTPUT.

The *SourceNode* of *References* of this type shall be a subtype of *CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* or an *Object* of the *ObjectType CtrlFunctionBlockType* or one of its subtypes.

### 4.7.4    HasInOutVar

This *ReferenceType* is a subtype of the *HasComponent ReferenceType* defined in UA Part 5. Its representation in the *AddressSpace* is specified in Table 21.

**Table 21 – HasInOutVar ReferenceType**

| Attributes | Value | | |
|---|---|---|---|
| BrowseName | HasInOutVar | | |
| InverseName | InOutVarOf | | |
| Symmetric | False | | |
| IsAbstract | False | | |
| **References** | **NodeClass** | **BrowseName** | **Comment** |
| Subtype of HasComponent Reference*Type* defined in [UA Part 5] | | | |

The *HasInOutVar ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference components of a *Ctrl Program Organization Unit* declared with the key word VAR_INOUTPUT.

The *SourceNode* of *References* of this type shall be a subtype of *CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* or an *Object* of the ObjectType *CtrlFunctionBlockType* or one of its subtypes.

### 4.7.5    HasLocalVar

This *ReferenceType* is a subtype of the *HasComponent ReferenceType* defined in UA Part 5. Its representation in the *AddressSpace* is specified in Table 22.

**Table 22 – HasLocalVar ReferenceType**

| Attributes | Value | | |
|---|---|---|---|
| BrowseName | HasLocalVar | | |
| InverseName | LocalVarOf | | |
| Symmetric | False | | |
| IsAbstract | False | | |
| **References** | **NodeClass** | **BrowseName** | **Comment** |
| Subtype of HasComponent Reference*Type* defined in [UA Part 5] | | | |

The *HasLocalVar ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference components of a *Ctrl Program Organization Unit* declared with the key word VAR.

The *SourceNode* of *References* of this type shall be a subtype of *CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* or an *Object* of the *ObjectType CtrlFunctionBlockType* or one of its subtypes.

### 4.7.6    HasExternalVar

This *ReferenceType* is a subtype of the *Organizes ReferenceType* defined in UA Part 5. Its representation in the *AddressSpace* is specified in Table 23.

**Table 23 – HasExternalVar ReferenceType**

| Attributes | Value | | |
|---|---|---|---|
| BrowseName | HasExternalVar | | |
| InverseName | ExternalVarOf | | |
| Symmetric | False | | |
| IsAbstract | False | | |
| **References** | **NodeClass** | **BrowseName** | **Comment** |
| Subtype of Organizes Reference*Type* defined in [UA Part 5] | | | |

The *HasExternalVar ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference components of a *Ctrl Program Organization Unit* declared with the key word VAR_EXTERNAL.

The *SourceNode* of *References* of this type shall be a subtype of *CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be a *Variable* or an *Object* of the *ObjectType CtrlFunctionBlockType* or one of its subtypes.

### 4.7.7    With

This *ReferenceType* is a subtype of the *NonHierarchicalReferences ReferenceType* defined in UA Part 5. Its representation in the *AddressSpace* is specified in Table 24.

**Table 24 – With ReferenceType**

| Attributes | Value | | |
|---|---|---|---|
| BrowseName | With | | |
| InverseName | Executes | | |
| Symmetric | False | | |
| IsAbstract | False | | |
| **References** | **NodeClass** | **BrowseName** | **Comment** |
| Subtype of NonHierarchicalReferences Reference*Type* defined in [UA Part 5] | | | |

The *With ReferenceType* is a concrete *ReferenceType* and can be used directly.

The semantic of this *ReferenceType* is to reference the *Ctrl Task* that executes a *Ctrl Program Organization Unit*.

The *SourceNode* of *References* of this type shall be an Object of the *ObjectType CtrlProgramOrganizationUnitType* or an instance of one of its subtypes.

The *TargetNode* of this *ReferenceType* shall be an *Object* of the *ObjectType CtrlTaskType* or one of its subtypes.

## 5   Definition of Ctrl Variable Attributes and Properties

### 5.1   Common Attributes

The common *Attributes* of *OPC UA Address Space Nodes* and their mapping from IEC 61131-3 are defined in Table 25.

**Table 25 – Common Node Attributes**

| Attribute | Use | DataType | Description |
|---|---|---|---|
| NodeId | Mandatory | NodeId | The *NodeId* is a unique identifier for a *Node* in an *OPC UA Address Space*. The identifier is server specific and its format is not defined in this specification. |
| NodeClass | Mandatory | NodeClass | The *NodeClass* is Variable for all *Ctrl Variables*. |
| BrowseName | Mandatory | QualifiedName | The *BrowseName* is a *QualifiedName* composed of a name string and a namespace index. It is used to create paths that can be passed to the *TranslateBrowsePathsToNodeIds Service* to get the *NodeId* of a *Variable Node*. This is typically used to get the NodeId of Variable in an *Object* instance based on the path known from the *Object* type. The *BrowseName* is not used to display the name of the *Node*.<br>The name part is generated from the *Ctrl Variable* name.<br>The namespace part depends of the scope where the Variable is defined. Chapter 7.3 describes the handling of namespaces. |
| DisplayName | Mandatory | LocalizedText | The *DisplayName* is a LocalizedText used by clients to display the name of a Node. It is composed of a localized text part and a LocaleId identifying the language of the text.<br>The *DisplayName* is server specific if the server supports localization of *Variable* names.<br>The *DisplayName* is composed of the *Ctrl Variable* name and an empty *LocaleId* string if the server does not support localization. |
| Description | Optional | LocalizedText | The optional Description shall describe the meaning of the Node using a localized text.<br>The *Description* may correspond to the element *Documentation* of the element *Variable* in PLCopen XML. |
| WriteMask | Optional | UInt32 | The *WriteMask* provides the optional information which attributes of the *Node* can be written by a client. This excludes the *Value Attribute* where the access is described by the *AccessLevel Attribute*.<br>The value of this Attribute is server specific.<br>Servers only supporting the use cases *Observation* and *Operation* are typically setting this *Attribute* to 0 or are not providing this optional *Attribute*.<br>Servers supporting also the use cases *Engineering* and *Service* may allow clients to change *Node Attributes*. |
| UserWriteMask | Optional | UInt32 | The user specific settings for the *WriteMask*. |

### 5.2   DataType

#### 5.2.1   Mapping of elementary data types

The mapping of IEC 61131-3 elementary data types to OPC UA data types is formally defined in Table 26. The OPC UA built in data types are used for the wire representation of the data type. Additional PLCopen specific OPC UA data type definitions are used to provide the special semantic if necessary.

**Table 26 – Mapping IEC 61131-3 elementary data types to OPC UA built in data types**

| IEC 61131-3 elementary data types | OPC UA built in data types | PLCopen specific OPC UA simple data type definitions | Comment |
|---|---|---|---|
| BOOL | Boolean | - | A one bit value (true or false). |
| SINT | SByte | - | An 8 bit signed integer value. |
| USINT | Byte | - | An 8 bit unsigned integer value. |
| INT | Int16 | - | A 16 bit signed integer value. |
| UINT | UInt16 | - | A 16 bit unsigned integer value. |
| DINT | Int32 | - | A 32 bit signed integer value. |
| UDINT | UInt32 | - | A 32 bit unsigned integer value. |
| LINT | Int64 | - | A 64 bit signed integer value. |
| ULINT | UInt64 | - | A 64 bit unsigned integer value. |
| BYTE | Byte | BYTE | The PLC open specific OPC UA simple data type BYTE is derived from the built in data type Byte. It describes that the type is used as bit string of length 8. |
| WORD | UInt16 | WORD | The PLC open specific OPC UA simple data type WORD is derived from the built in data type UInt16. It describes that the type is used as bit string of length 16 |
| DWORD | UInt32 | DWORD | The PLC open specific OPC UA simple data type DWORD is derived from the built in data type UInt32. It describes that the type is used as bit string of length 32 |
| LWORD | UInt64 | LWORD | The PLC open specific OPC UA simple data type LWORD is derived from the built in data type UInt64. It describes that the type is used as bit string of length 64 |
| REAL | Float | - | OPC UA definition: An IEEE-754 single precision (32 bit) floating point value.<br>IEC 61131-3 definition: Real (32 bit) with a range of values as defined in IEC 60559 for the basic single width floating-point format.<br>Both standards are identical. |
| LREAL | Double | - | OPC UA definition: An IEEE-754 double precision (64 bit) floating point value.<br>IEC 61131-3 definition: Long real (64 bit) with a range of values as defined in IEC 60559 for the basic double width floating-point format.<br>Both standards are identical. |
| STRING | String | STRING | The PLC open specific OPC UA simple data type STRING is derived from the built in data type String. It describes that the type is used as a variable-length single-byte character string. |
| CHAR | Byte | CHAR | The PLC open specific OPC UA simple data type CHAR is derived from the built in data type Byte. It describes that the type is used as single-byte character |
| WSTRING | String | - | OPC UA definition: A sequence of UTF8 characters.<br>IEC 61131-3 definition: Variable-length double-byte character string |
| WCHAR | UInt16 | WCHAR | The PLC open specific OPC UA simple data type WCHAR is derived from the built in data type UInt16. It describes that the type is used as double-byte character. |
| DT DATE_AND_TIME | DateTime | - | OPC UA definition: A 64-bit signed integer which represents the number of 100 nanosecond intervals since January 1, 1601.<br>IEC 61131-3 definition: Date and time of day. |
| DATE | DateTime | DATE | The PLC open specific OPC UA simple data type DATE is derived from the built in data type DateTime. It describes that the type is used as a date only. |
| TOD TIME_OF_DAY | DateTime | TOD | The PLC open specific OPC UA simple data type TOD is derived from the built in data type DateTime. It describes that the type is used as time of day only. |
| TIME | Double | Duration (defined by OPC UA) | The OPC UA simple data type Duration is derived from the built in data type Double. It describes that the type is used as interval of time in milliseconds. |

### 5.2.2  Mapping of generic data types

The mapping of IEC 61131-3 generic data types to OPC UA data types is formally defined in Table 27. Since the generic data type should not be used in user-declared Ctrl Program

Organization Units, this mapping definition is defined for completeness but is normally not used in an OPC UA AddressSpace.

**Table 27 – Mapping IEC 61131-3 generic data types to OPC UA data types**

| IEC 61131-3 generic data types | OPC UA data types | Description |
|---|---|---|
| ANY | BaseDataType | This abstract OPC UA *DataType* defines a value that can have any valid OPC UA *DataType*. |
| ANY_DERIVED | BaseDataType | |
| ANY_ELEMENTARY | BaseDataType | |
| ANY_MAGNITUDE | BaseDataType | |
| ANY_NUM | Number | This abstract OPC UA *DataType* defines a number value that can have any of the OPC UA Number subtypes. |
| ANY_REAL | Number | |
| ANY_INT | Number | |
| ANY_BIT | Number | |
| ANY_STRING | String | This OPC UA Built-in DataType defines a Unicode character string that should exclude control characters that are not whitespaces (0x00 - 0x08, 0x0E-0x1F or 0x7F). |
| ANY_DATE | DateTime | This OPC UA *Built-in DataType* defines a Gregorian calendar date. It is a 64-bit signed integer which represents the number of 100 nanosecond intervals since January 1, 1601. |

### 5.2.3 Mapping of derived data types

### 5.2.3.1 Mapping of enumerated data types

Both OPC UA and IEC 61131-3 allow the definition of enumerations on a data type or on a variable instance.

In OPC UA the enumerated data types are defined as subtypes of Enumeration. The data has an *EnumStrings Property* that contains the possible string values. The value is transferred as integer on the wire where the integer defines the index into the *EnumStrings* array. The index is zero based and has no gaps. Another option is to provide the possible string values in the *Property EnumValues*. This option is used if individual integer values are assigned to the string. The used option depends on the way the string enumeration is defined in the *Controller* program. If integer values are assigned to the string values the *Property EnumValues* is used to represent the enumeration values. If the integer value is zero based and has no gaps the *EnumStrings Property* should be used since the processing on the client side is more efficient.

The definition on a variable instance is using the MultiStateDiscreteType Variable Type which defines also the *EnumStrings* or the *EnumValues Property* containing the enumeration values as string array.

Example for an enumerated data type declaration in IEC 61131-3:

```
TYPE
  ANALOG_SIGNAL_TYPE : (SINGLE_ENDED, DIFFERENTIAL) ;
END_TYPE
```

Example for use of an enumeration in a *Ctrl Variable* instantiation in IEC 61131-3:

```
VAR
  Y : (Red, Yellow, Green) ;
END_VAR
```

The IEC 61131-3 enumeration data type declaration is mapped to an OPC UA enumeration data type. The representation in the address space is formally defined in Table 28.

**Table 28 – Enumeration Data Type Definition**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | <IEC Data Type Name> | | | | |
| IsAbstract | False | | | | |
| References | NodeClass | BrowseName | DataType | TypeDefinition | ModellingRule |
| Subtype of the Enumeration data type defined in [UA Part 5] | | | | | |
| HasProperty | Variable | EnumString | String [ ] | PropertyType | Optional |
| HasProperty | Variable | EnumValues | EnumValueDataType [ ] | PropertyType | Optional |

The Property *EnumString* is defined in [UA Part 5].

The *Property EnumValues* is defined in [UA Part 5].

The IEC 61131-3 enumeration in a *Ctrl Variable* declaration is mapped to a MultiStateDiscreteType Variable Type defined in [UA Part 8].

### 5.2.3.2  Mapping of subrange data types

IEC 61131-3 defines the subrange for all integer data types (ANY_INT) which excludes real values.

OPC UA has no standard concept to limit the range on the data type.

Example for a subrange data type declaration in IEC 61131-3:

```
TYPE
  ANALOG_DATA : INT (-4095..4095) ;
END_TYPE
```

Example for use of a subrange in a *Ctrl Variable* instantiation in IEC 61131-3:

```
VAR
  Z : SINT (5..95) ;
END_VAR
```

The IEC 61131-3 subrange is mapped to two OPC UA properties formally defined in Table 29.

**Table 29 – Subrange Property Definition**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | <Ctrl Variable Name> or <IEC Data Type Name> | | | | |
| References | NodeClass | BrowseName | DataType | TypeDefinition | ModellingRule |
| Instance of any Variable Type or a Data Type Node. | | | | | |
| HasProperty | Variable | SubrangeMin | Number | PropertyType | Mandatory |
| HasProperty | Variable | SubrangeMax | Number | PropertyType | Mandatory |

The Property SubrangeMin contains the lower bound of the subrange. The data type depends on the elementary data type used for the subrange.

The Property SubrangeMax contains the upper bound of the subrange. The data type depends on the elementary data type used for the subrange.

The IEC 61131-3 subrange data type is mapped to an OPC UA number data type derived from the corresponding elementary data types defined in Table 26. The data type has the two Properties defined in Table 29. The IEC example in this chapter is mapped to an OPC UA data type with the name ANALOG_DATA which is a subtype of Int16.

The IEC 61131-3 subrange in a *Ctrl Variable* declaration is mapped to the two Properties defined in Table 29. The Properties are children of the OPC UA Variable representing the *Ctrl Variable*.

### 5.2.3.3  Mapping of array data types

OPC UA provides the information if a value is an array in the Variable Attributes ValueRank and ArrayDimensions. Every data type can be exposed as array. Arrays can have multiple dimensions. The dimension is defined through the Attribute ValueRank. Arrays can have variable or fixed lengths. The length of each dimension is defined by the Attribute ArrayDimensions. The array index starts with zero.

IEC 61131-3 allows the declaration of array data types with one or multiple dimensions and an index range instead of a length.

OPC UA has no standard concept for defining special array data types or exposing index ranges.

Example for an array data type declaration in IEC 61131-3:

```
TYPE
  ANALOG_16_INPUT_DATA : ARRAY [1..16] OF INT ;
END_TYPE
```

Example for use of an array in a *Ctrl Variable* instantiation in IEC 61131-3:

```
VAR
  MyArray : ARRAY [1..16] OF INT;
END_VAR
```

The IEC 61131-3 array data type is mapped to three OPC UA properties formally defined in Table 30.

**Table 30 – Array Data Type Property Definition**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | <Ctrl Variable Name> or <IEC Data Type Name> | | | | |
| References | NodeClass | BrowseName | DataType | TypeDefinition | ModellingRule |
| Instance of any Variable Type or a Data Type Node. | | | | | |
| HasProperty | Variable | Dimensions | UInt32 | PropertyType | Mandatory |
| HasProperty | Variable | IndexMin | UInt32 [ ] | PropertyType | Mandatory |
| HasProperty | Variable | IndexMax | UInt32 [ ] | PropertyType | Mandatory |

The Property Dimensions contains the number of dimensions of the array.

The Property IndexMin contains an array of lower bounds, one for each array dimension.

The Property IndexMax contains an array of upper bounds, one for each array dimension.

The IEC 61131-3 array data type is mapped to an OPC UA data type derived from the corresponding elementary data types defined in Table 26. The data type has the two Properties defined in Table 30. The IEC example in this chapter is mapped to an OPC UA data type with the name ANALOG_16_INPUT_DATA which is a subtype of Int16.

The IEC 61131-3 array in a *Ctrl Variable* declaration is mapped to the two Properties defined in Table 30. The Properties are children of the OPC UA Variable representing the *Ctrl Variable*.

### 5.2.3.4  Mapping of structure data types

IEC 61131-3 structure data types are mapped as subtypes of the OPC UA DataType Structure. OPC UA servers must explicitly describe how structured DataTypes are encoded / decoded and provide this information to the client which is using it while reading / writing structure data.

The following example of an IEC 61131-3 structure data type declaration (using Structured Text) will be used for further illustrations. This structure data type comprises three structure elements of different elementary data types.

```
TYPE ExampleIEC611313Structure:
  STRUCT
    IntStructureElement: INT;
    RealStructureElement: REAL;
    BoolStructureElement: BOOL;
  END_STRUCT;
END_TYPE
```

The following Figure 20 shows the mapping of the above example.
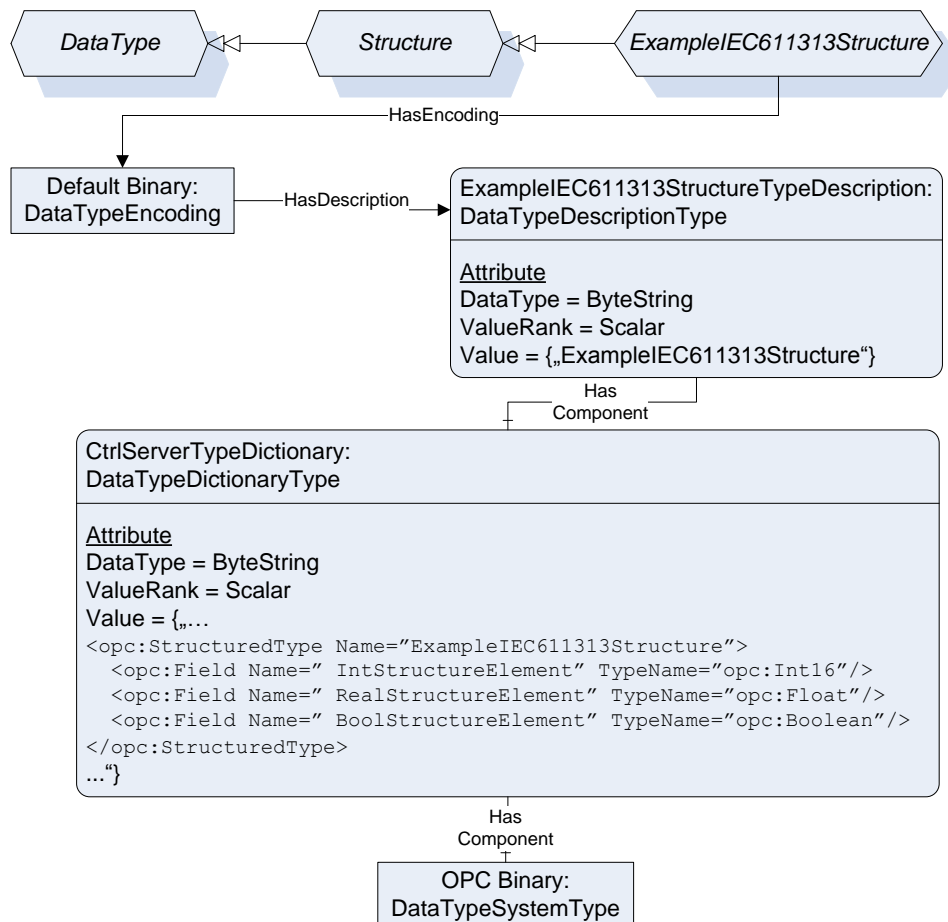


**Figure 20 – Mapping of structure data types**

Ctrl servers must support the binary encoding ("Default Binary"). Additionally, a XML encoding may be provided (not shown in above figure). A Server provides a DataTypeDictionary Variable describing all necessary DataTypes. Each DataType is represented by a DataTypeDescription Variable. Optionally, a Property DictionaryFragment may be available, allowing clients not to read the complete DataTypeDictionary in order to get the information about only a single DataType (not shown in above figure).

It is strongly recommended for Ctrl servers to provide additionally the structured data as a set of sub variables (components of the variable) providing the structure as several separated values. This allows clients that do not support complex data to access the scalar values. The following Figure 21 shows an example (instances based on the above type descriptions).
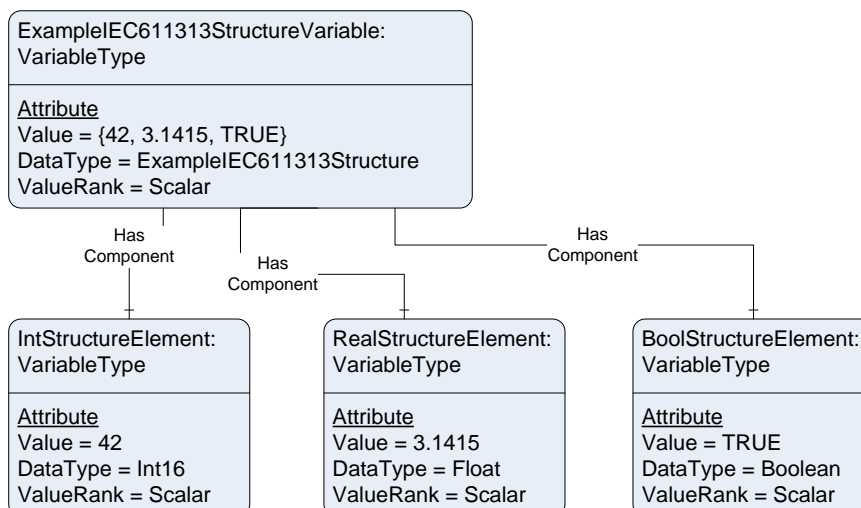
**Figure 21 – Mapping of structure data types to Variable components**

If a structure element is not an elementary data type, it has to be divided again into sub variables.

It is recommended that Ctrl servers do support complex data. If a server does not support complex data it provides only sub variables for structure variables. The structured variable would be a Folder object in this case.

### 5.3  Variable specific Node Attributes

### 5.3.1  General

The *Variable* specific *Attributes* of *OPC UA Address Space Nodes* and their mapping from IEC 61131-3 are defined in Table 31.

**Table 31 – Variable Node Attributes**

| Attribute | Use | DataType | Description |
|---|---|---|---|
| Value | Mandatory | Defined by DataType | The most recent value of the *Variable* that the server has. Its data type is defined by the *DataType, ValueRank* and *ArrayDimension* Attribute. |
| DataType | Mandatory | Nodeid | The *DataType* of the *Variable Value*. It defines the type specific content of the Value together with the *ValueRank* and the *ArrayDimension Attributes*, The mapping is defined in 5.2. |
| ValueRank | Mandatory | Int32 | This *Attribute* indicates whether the *Value* of the *Variable* is an array and how many dimensions the array has. *Ctrl Variables* declared as scalar type have the *ValueRank* -1. *Ctrl Variables* declared with the key word ARRAY…OF have a *ValueRank* that indicates the number of dimension of the array declared for the *Ctrl Variable*. |
| ArrayDimensions | Optional | UInt32[] | This *Attribute* specifies the length of each dimension for an array value. The *Attribute* is intended to describe the capability of the *Variable*, not the current size. The dimension entries have the length defined with the key word ARRAY…OF. |
| AccessLevel | Mandatory | Byte | The AccessLevel Attribute is used to indicate how the Value of a *Variable* can be accessed (read/write) and if it contains current and/or historic data. The handling of access to historic data is server specific and is not part of this specification. The mapping of the read and write access part of the *AccessLevel* is defined in 5.3.2 |
| UserAccessLevel | Mandatory | Byte | The user specific settings for the *AccessLevel*. |
| MinimumSampling Interval | Optional | Duration | The *MinimumSamplingInterval Attribute* indicates how "current" the *Value* of the *Variable* will be kept. It specifies (in milliseconds) how fast the server can reasonably sample the value for changes. A MinimumSamplingInterval of 0 indicates that the server is to monitor the item continuously. A MinimumSamplingInterval of -1 means indeterminate. The value of this Attribute is server specific. |
| Historizing | Mandatory | Boolean | Indicates if the server is currently collecting history for the *Variable Value*. The support of value history is server specific. |

### 5.3.2  Access Level

If the IEC attribute CONSTANT is set, the Access Level shall be read only.

The IEC standard does not define a key word to set the Access Level for a Ctrl Variable. The configuration in a programming system is vendor specific but it is recommended to provide a configuration option for the OPC UA Access Level.

When using the PLCopen XML format the AccessLevel shall be provided in PLCopen XML *Additional Data* in the XML element *addData* using the XML element UaAccessLevel as part of the XML element representing the variable. The value is a bit mask where the first bit indicates the read access and the second bit indicates the write access.

### 5.4  Variable Properties

### 5.4.1  IEC Ctrl Variable Keywords

The IEC 61131-3 key word mapping to OPC UA *Properties* is formally defined in Table 32.

**Table 32 – IEC 61131-3 Variable Key Word Property Definition**

| Attribute | Value | | | | |
|---|---|---|---|---|---|
| BrowseName | <Ctrl Variable Name> | | | | |
| References | NodeClass | BrowseName | DataType | TypeDefinition | ModellingRule |
| Instance of any Variable Type. | | | | | |
| HasProperty | Variable | RETAIN | Boolean | PropertyType | Optional |
| HasProperty | Variable | NON_RETAIN | Boolean | PropertyType | Optional |
| HasProperty | Variable | CONSTANT | Boolean | PropertyType | Optional |
| HasProperty | Variable | AT | String | PropertyType | Optional |

The *Property* RETAIN indicates if the RETAIN key word is set for the *Ctrl Variable*. It provides an explicit declaration of "warm start" behaviour of the *Ctrl Variable* (and *Ctrl Function Blocks* and *Ctrl Programs*).

The *Property* NON_RETAIN indicates if the NON_RETAIN key word is set for the *Ctrl Variable*. It provides an explicit declaration of "warm start" behaviour of the *Ctrl Variable* (and *Ctrl Function Blocks* and *Ctrl Programs*).

The *Property* CONSTANT indicates if the CONSTANT key word is set for the *Ctrl Variable*. It provides a declaration of a fixed value for the *Ctrl Variable*. The *Ctrl Variable* cannot be modified.

The *Property* AT contains the location assignment to the *Ctrl Variable* as string if the AT key word is set for the *Ctrl Variable*.

### 5.4.2  Configuration of OPC UA defined Properties

The IEC standard does not define key words to configure information like the value range or the engineering unit for a *Ctrl Variable*. The configuration in a programming system is vendor specific but this specification defines the export format in the PLCopen XML *Additional Data* in the XML element *addData*.

The *InstrumentRange Property* defined in UA Part 8 shall be provided in the XML element *UaInstrumentRange* as part of the XML element representing the *Ctrl Variable*. The attributes of the XML element are formally defined in Table 33.

**Table 33 – Range XML attributes**

| Name | Type | Use | Default |
|------|------|-----|---------|
| Low | double | required | |
| High | double | required | |

The *EURange Property* defined in UA Part 8 shall be provided in the XML element *UaEURange* as part of the XML element representing the *Ctrl Variable*. The attributes of the XML element are formally defined in Table 33.

The *EngineeringUnits Property* defined in UA Part 8 shall be provided in the XML element *UaEngineeringUnits* as part of the XML element representing the *Ctrl Variable*.

## 6    Objects used to organise the AddressSpace structure

### 6.1    DeviceSet as entry point for engineering applications (Mandatory)

The full object component hierarchy based on *Object Types* defined in this specification shall be provided as components of the *DeviceSet Object* defined in UA Part DI. Figure 22 provides an example for such a component hierarchy.

The *DeviceSet Object* is typically used as entry point by a UA client in the use cases *Engineering* and *Service*.
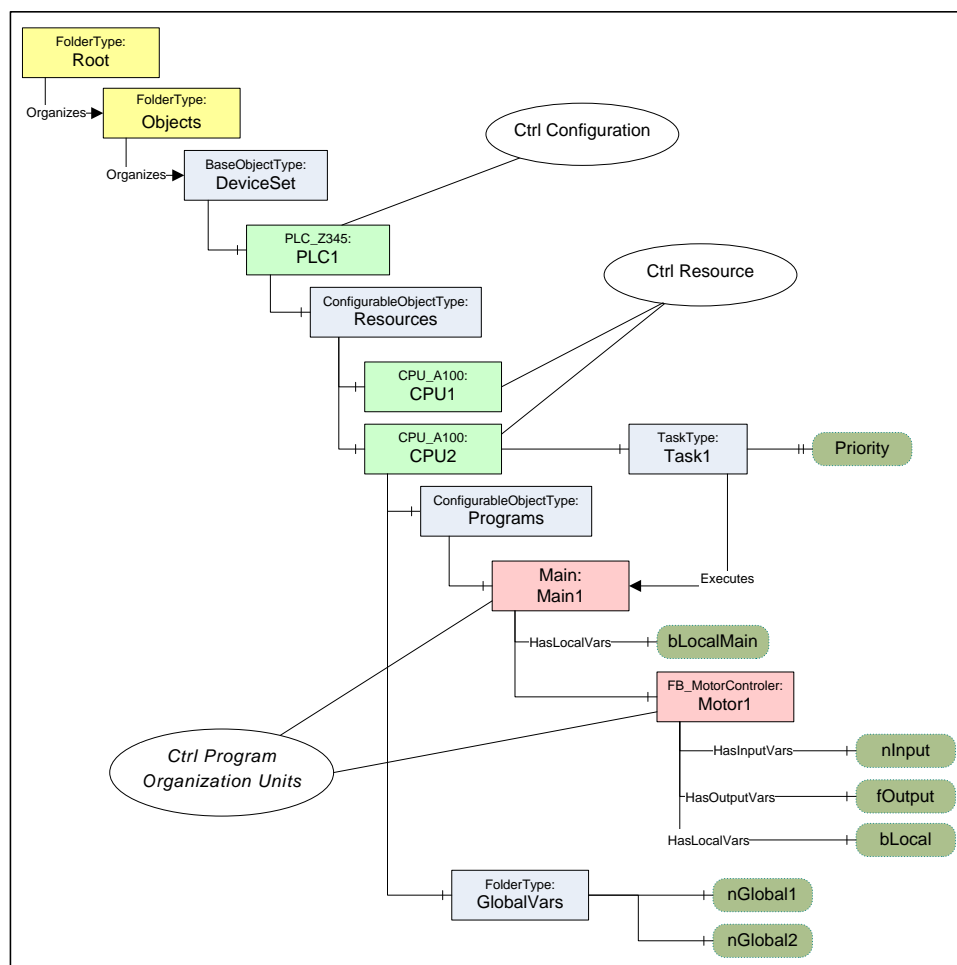


**Figure 22 – DeviceSet as entry point for engineering applications**

### 6.2    CtrlTypes Folder for server specific Object Types (Mandatory)

The server specific *ObjectTypes* like vendor specific *Ctrl Configuration* types or user specific *Ctrl Function Block* types can be found by a UA client by following the type hierarchy.

To provide UA clients all relevant server specific types in one place, the *Ctrl Function Block* types shall be referenced directly or indirectly from the *CtrlTypes Folder Object* using *Organizes References*. Other types like *Ctrl Resources* or *Ctrl Program* types may be included in addition. The CtrlTypes node is formally defined in Table 34

**Table 34 – CtrlTypes definition**

| Attribute | Value | | | |
|---|---|---|---|---|
| BrowseName | CtrlTypes | | | |
| **References** | **NodeClass** | **BrowseName** | **TypeDefinition** | **Description** |
| Organized by the ObjectType Folder defined in UA Part 5 | | | | |
| HasTypeDefinition | ObjectType | Folder | | |
| Organizes | Object | \<Server specific\> | FolderType | Optional server specific additional structuring of the type information building to a type catalogue |
| Organizes | ObjectType | \<Server specific\> | | Server specific Object Types |

The server may provide additional Folder objects below the *CtrlTypes Object* to organize the types. This can be used to create a library structure like in the example in Figure 23.
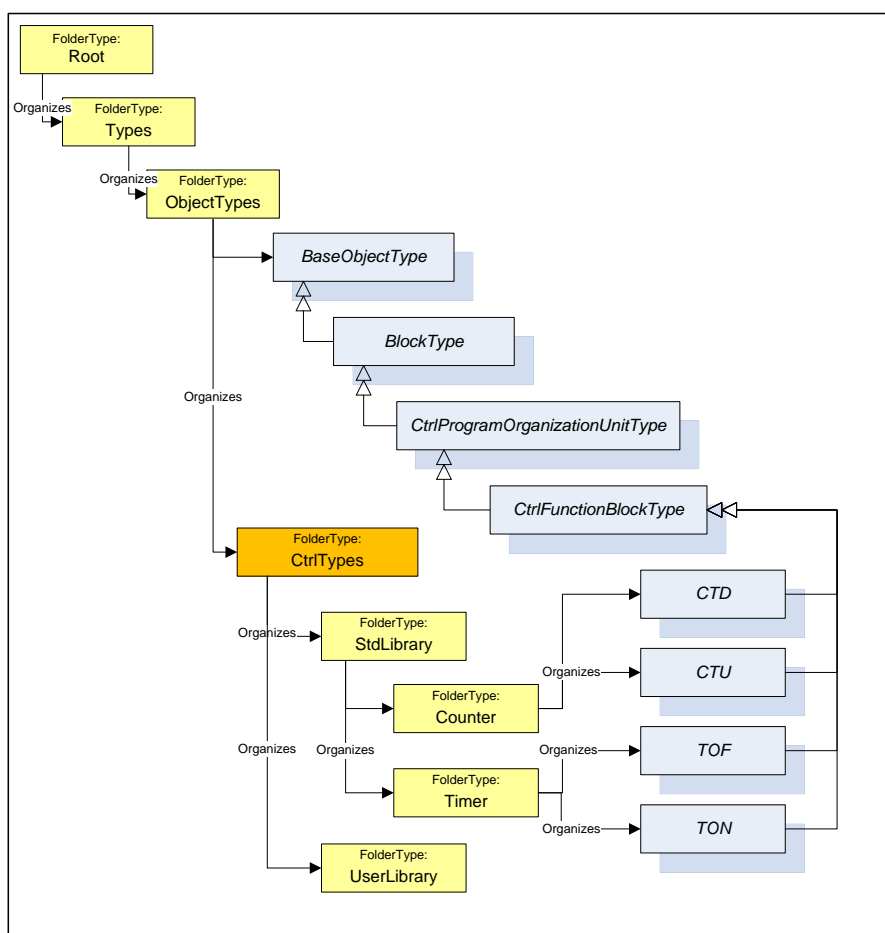


**Figure 23 – CtrlTypes Folder used to structure POU types**

## 6.3  Entry point for Observation and Operation (Examples)

The entry point for UA client for the use cases *Observation* and *Operation* is the *Objects Folder*. One typical entry point is a list of *Objects* representing *Ctrl Resources*. Additional *Folders Objects* used to structure the *Ctrl Resources* into a hierarchy are server specific. Such an example is shown in Figure 24.
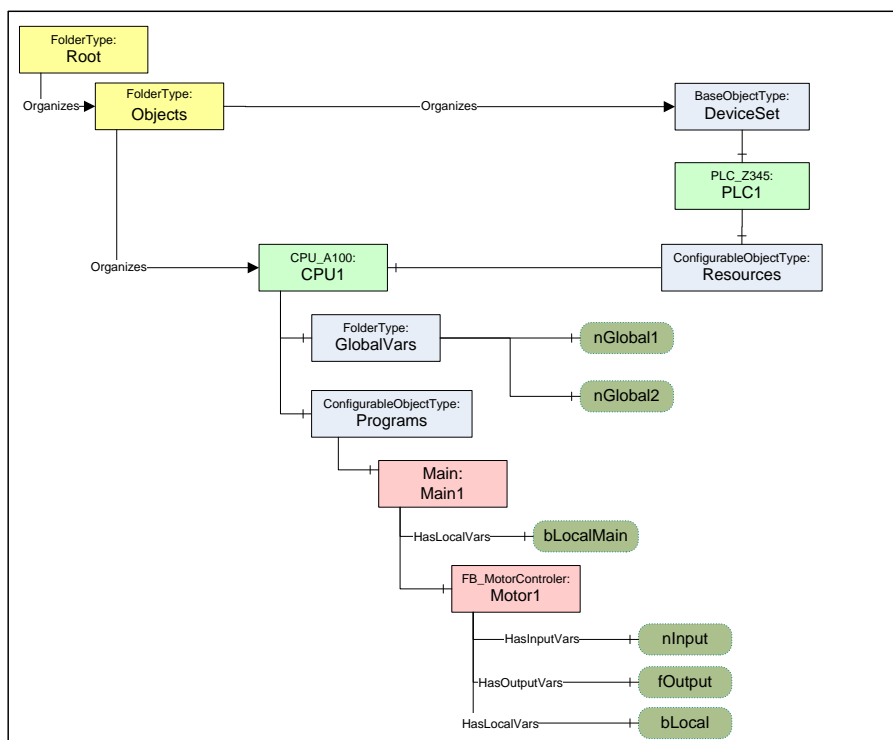
**Figure 24 – Browse entry point for Operation with Ctrl Resource**

Servers that want to hide some of the components of a *Ctrl Resources* can create a *Folder Object* representing the *Ctrl Resources* and can use *Organizes References* to reference only the components of the *Ctrl Resources* that should be visible in this part of the hierarchy. Such an example is shown in Figure 25.
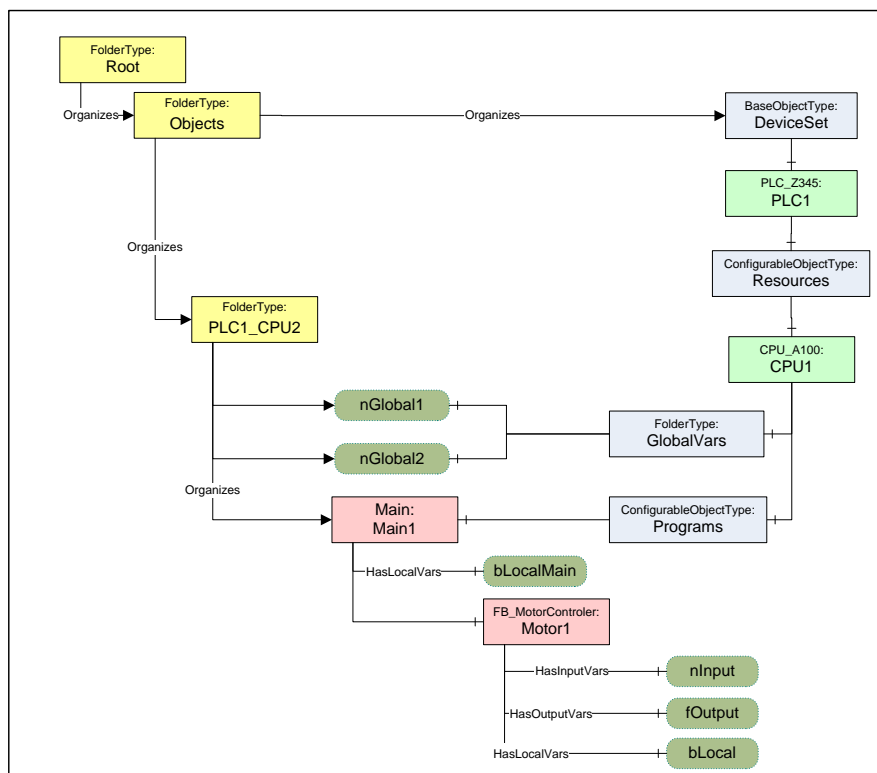


**Figure 25 – Browse entry point for Operation with simplified Folder**

## 7    System Architecture and Profiles

### 7.1    System Architecture

#### 7.1.1    General

This chapter describes typical system architectures where this specification can be applied. Figure 26 shows a possible configuration where OPC UA based interfaces are involved.
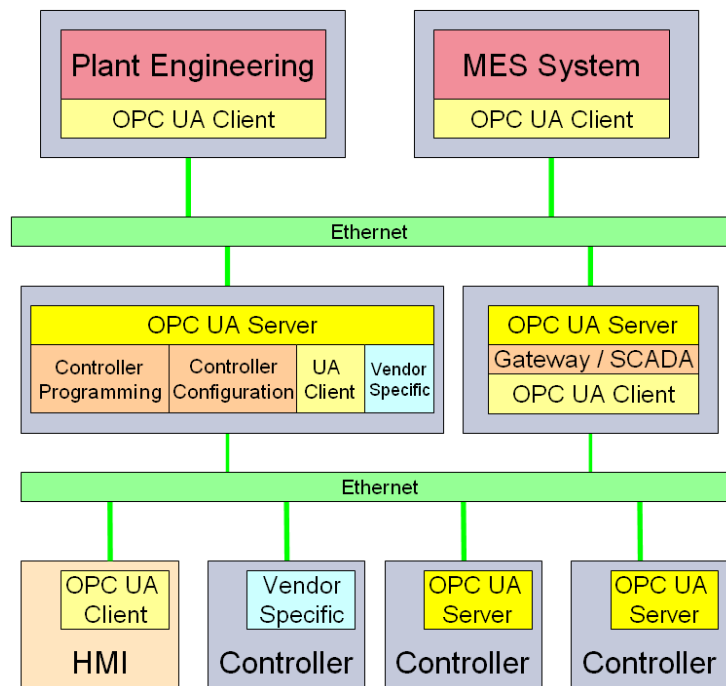


**Figure 26 – System Architecture**

#### 7.1.2    Embedded OPC UA Server

Embedded OPC UA servers are directly integrated into a *Controller* providing *Ctrl Program* and *Ctrl Function Block Objects*. Such a server allows direct access to information from a *Controller* using the OPC UA protocol on the wire. Other embedded applications like HMI acting as OPC UA clients can access the information from *Controllers* directly without the need of a PC.

#### 7.1.3    PC based OPC UA Server

OPC UA servers running on a PC platform are capable of providing access to multiple *Controllers*.  They are providing full type information for *Ctrl Resource*, *Ctrl Program and Ctrl Function Block Objects*. The communication to the *Controllers* may use OPC UA or a proprietary protocol on the wire.

#### 7.1.4    PC based OPC UA Server with engineering capabilities

In addition to PC based OPC UA servers, this type of server includes access to the engineering system for the *Controllers* allowing access to the configuration for the use cases *Engineering* and *Service*.

### 7.2    Conformance Units and Profiles

This chapter defines the corresponding profiles and conformance units for the OPC UA Information Model for IEC 61131-3. Profiles are named groupings of conformance units.

Facets are profiles that will be combined with other *Profiles* to define the complete functionality of an OPC UA *Server* or *Client.* The following tables specify the facets available for *Servers* that implement the IEC 61131-3 Information Model companion standard.

Table 35 describes *Conformance Units* included in the minimum needed facet. It requires the support for profile *BaseDevice Server Facet* defined in [UA Part DI]. It is used together with the *Embedded UA Server* profile or the *Standard UA Server* profile defined in [UA Part 7].

A server supporting all data types including complex data types must support the *ComplexType Server Facet* defined in [UA Part 7].

**Table 35 – *Controller Operation Server Facet* Definition**

| Conformance Unit | Description | Optional/ Mandatory |
|---|---|---|
| Ctrl DeviceSet | Support the full component hirarchy with *Ctrl Configuration*, *Ctrl Resource*, *Ctrl Program* and *Ctrl FunctionBlock* below the DeviceSet Object defined in [UA Part DI]. | M |
| Ctrl Configuration | Support vendor defined *Ctrl Configuration* object types and object instances. | M |
| Ctrl Resource | Support vendor defined *Ctrl Resource* object types and object instances | M |
| Ctrl Program | Support user defined *Ctrl Program* object types and object instances. | M |
| Ctrl FunctionBlock | Support user defined *Ctrl FunctionBlock* object types and object instances. | M |
| Ctrl Task | Support of *Ctrl Task* objects. | O |
| Ctrl References | Support of reference types specified in the IEC 61131-3 Information Model companion standard. | O |
| This profile requires the support of the profile *BaseDevice_Server_*Facet defined in [UA Part DI] | | |

Table 36 defines a facet for the support of the engineering information defined in the IEC 61131-3 Information Model. The *Controller Engineering Server Facet* requires the *Controller Operation Server Facet*.

**Table 36 – *Controller Engineering Server Facet* Definition**

| Conformance Unit | Description | Optional/ Mandatory |
|---|---|---|
| Ctrl Engineering Data | Support to provide all engineering data defined in this specification like properties describing data types. | M |
| Ctrl Engineering Change | Support of engineering data changes through OPC UA | O |
| Ctrl Type Creation | Support of type node creation through *NodeManagement* Services to create *Ctrl Program Organization Unit* declarations. | O |

Table 37 defines a facet available for *Clients* that implement the IEC 61131-3 Information Model standard. Servers implementing the *Controller Engineering Server Facet* may use this facet to restrict the engineering features to clients supporting this *Client* facet.

**Table 37 – *Controller Engineering Client Facet* Definition**

| Conformance Unit | Description | Optional/ Mandatory |
|---|---|---|
| Ctrl Client Information Model | Consume objects that conform to the types specified in the IEC 61131-3 Information Model companion standard. | M |
| Ctrl Client Engineering Data | Consume engineering data defined in the IEC 61131-3 Information Model companion standard like properties describing data types. | M |
| Ctrl Client Engineering Change | Use engineering data changes through OPC UA | O |
| Ctrl Type Creation | Use type node creation through *NodeManagement* Services to create *Ctrl Program Organization Unit* declarations. | O |

## 7.3  Handling of OPC UA namespaces

Namespaces are used by OPC UA to create unique identifiers across different naming authorities. The *Attributes NodeId* and *BrowseName* are identifiers. A node in the UA *Address Space* is unambiguously identified using a *NodeId*. Unlike *NodeIds*, the *BrowseName* cannot

be used to unambiguously identify a node. Different nodes may have the same *BrowseName*. They are used to build a browse path between two nodes or to define a standard *Property*.

Servers may often choose to use the same namespace for the *NodeId* and the *BrowseName*. However, if they want to provide a standard *Property*, its *BrowseName* shall have the namespace of the standards body although the namespace of the *NodeId* reflects something else, for example the local server. Another example shown in Figure 27 is the *ParameterSet* and the *GlobalVars* object components of a *Ctrl Resource* instance. The *ParameterSet* node *BrowseName* shall use the OPC DI namespace and the *GlobalVars* node *BrowsName* shall use the namespace defined by this specification. All *NodeIds* of nodes not defined in this specification shall not use the standard namespaces and are typically using the same namespace like the *Ctrl Resource* object instance, for example local server.

Table 38 provides a list of mandatory and optional namespaces used in a *Controller Server*.

**Table 38 – Namespaces used in a Controller Server**

| Namespace | Description | Use |
|---|---|---|
| http://opcfoundation.org/UA/ | Namespace for *NodeIds* and *BrowsNames* defined in the OPC UA specification. This namespace shall have namespace index 0. | Mandatory |
| Local Server URI | Namespace for nodes defined in the local server. This may include types and instances used in a *Ctrl Resource* represented by the server. This namespace shall have namespace index 1. | Mandatory |
| http://opcfoundation.org/UA/DI/ | Namespace for *NodeIds* and *BrowsNames* defined in [UA Part DI]. The namespace index is server specific. | Mandatory |
| http://PLCopen.org/OpcUa/IEC61131-3/ | Namespace for *NodeIds* and *BrowsNames* defined in this specification. The namespace index is server specific. | Mandatory |
| http://PLCopen.org/OpcUa/IEC61131-3/FB/ | A server may provide IEC or PLCopen defined *Ctrl Function Block* libraries. | Optional |
| User defined types and instances in a Ctrl Resource | A server that provides access to different *Ctrl Resources* may provide a separate namespace for each *Ctrl Resources* if it is required to create unique identifiers across *Ctrl Resources*. | Optional |
| Vendor specific types | A server may provide vendor specific types like types derived from *Ctrl Configuration* or *Ctrl Resource* in a vendor specific namespace. | Optional |
| Global user defined library | A server may provide global user defined *Ctrl Function Block* libraries in a user specific namespace. | Optional |

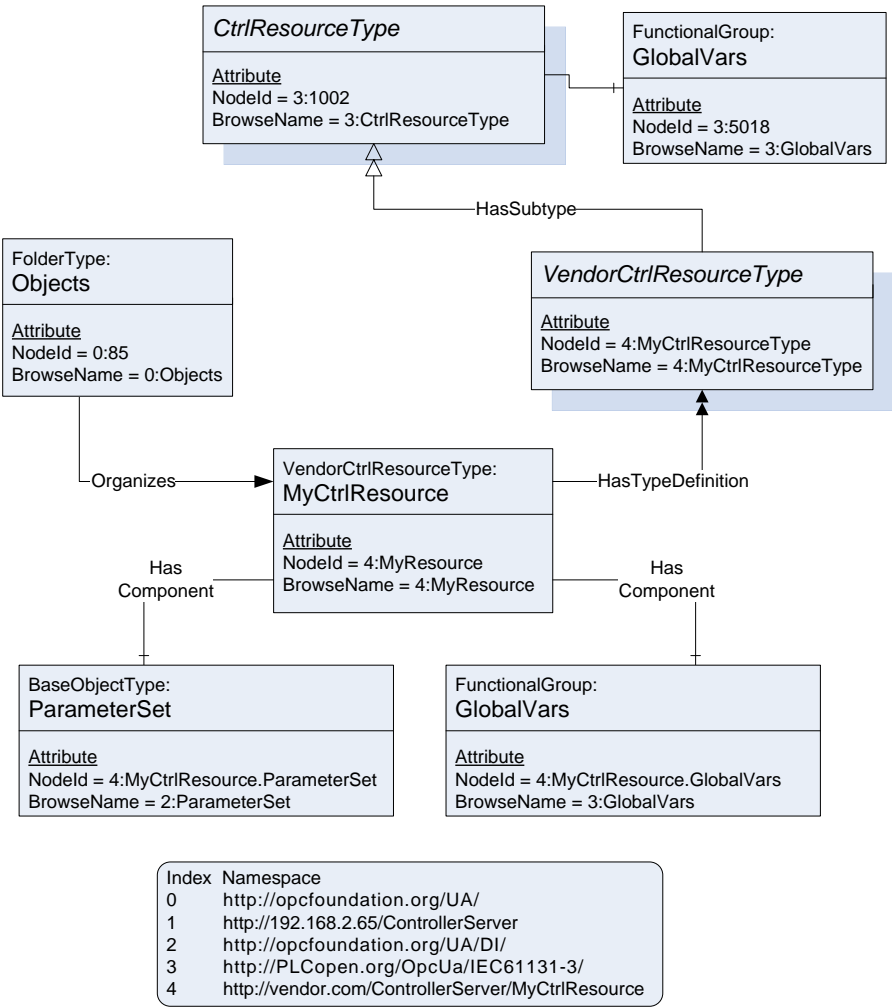Figure 27 shows an example for the use of namespaces in NodeIds and BrowseNames.

**Figure 27 – Example for the use of namespaces in NodeIds and BrowseNames**

## Annex A (normative): IEC 61131-3 Namespace and Mappings

### A.1    Namespace and identifiers for IEC61131-3 Information Model

The namespace for all nodes defined in this document is "http://PLCopen.org/OpcUa/IEC61131-3/". The *NodeIds* for the defined nodes are composed of this namespace and a numeric identifier for the *Node*. Table 39 specifies the mapping of object types, instance declarations, objects and reference types to numeric identifiers.

**Table 39 – Numeric Identifiers for IEC 61131-3 defined nodes**

| Node | Numeric Identifier |
| --- | --- |
| **Object Types** | |
| CtrlConfigurationType | 1001 |
| CtrlResourceType | 1002 |
| CtrlProgramOrganizationUnitType | 1003 |
| CtrlProgramType | 1004 |
| CtrlFunctionBlockType | 1005 |
| CtrlTaskType | 1006 |
| SFCType | 1007 |
| | |
| **Reference Types** | |
| HasInputVar | 4001 |
| HasOutputVar | 4002 |
| HasInOutVar | 4003 |
| HasLocalVar | 4004 |
| HasExternalVar | 4005 |
| With | 4006 |
| | |
| **Objects** | |
| CtrlConfigurationType_ParameterSet | 5001 |
| CtrlConfigurationType_MethodSet | 5002 |
| CtrlConfigurationType_Identification | 5003 |
| CtrlConfigurationType_Resources | 5004 |
| CtrlConfigurationType_Resources_SupportedTypes | 5005 |
| CtrlConfigurationType_GlobalVars | 5006 |
| CtrlConfigurationType_AccessVars | 5007 |
| CtrlConfigurationType_ConfigVars | 5008 |
| CtrlConfigurationType_Configuration | 5009 |
| CtrlConfigurationType_Diagnostic | 5010 |
| CtrlResourceType_ParameterSet | 5011 |
| CtrlResourceType_MethodSet | 5012 |
| CtrlResourceType_Identification | 5013 |
| CtrlResourceType_Tasks | 5014 |
| CtrlResourceType_Tasks_SupportedTypes | 5015 |
| CtrlResourceType_Programs | 5016 |
| CtrlResourceType_Programs_SupportedTypes | 5017 |
| CtrlResourceType_GlobalVars | 5018 |
| CtrlResourceType_Configuration | 5019 |
| CtrlResourceType_Diagnostic | 5020 |
| CtrlProgramOrganizationUnitType_ParameterSet | 5021 |
| CtrlProgramOrganizationUnitType_MethodSet | 5022 |
| CtrlProgramOrganizationUnitType_Identification | 5023 |
| CtrlProgramType_ParameterSet | 5024 |
| CtrlProgramType_MethodSet | 5025 |
| CtrlProgramType_Identification | 5026 |
| CtrlFunctionBlockType_ParameterSet | 5027 |
| CtrlFunctionBlockType_MethodSet | 5028 |
| CtrlFunctionBlockType_Identification | 5029 |
| | |
| **Variables** | |
| CtrlProgramOrganizationUnitType_Body | 6001 |
| CtrlProgramType_Program | 6002 |
| CtrlFunctionBlockType_FunctionBlock | 6003 |

| CtrlTaskType_Priority | 6004 |
|---|---|
| CtrlTaskType_Interval | 6005 |
| CtrlTaskType_Single | 6006 |
|  |  |
| **Methods** |  |
| CtrlConfigurationType_MethodSet_Start | 7001 |
| CtrlConfigurationType_MethodSet_Stop | 7002 |
| CtrlResourceType_MethodSet_Start | 7003 |
| CtrlResourceType_MethodSet_Stop | 7004 |
|  |  |

The CSV file containing the numeric identifiers for this namespace can be found here:

http://www.PLCopen.org/OpcUa/IEC61131-3/NodeIds.csv

## A.2    Profile URIs for IEC61131-3 Information Model

Table 40 defines the Profile URIs for the IEC 61131-3 Information Model companion standard.

**Table 40 – Profile URIs**

| Profile | Profile URI |
|---|---|
| Controller Operation Server Facet | http://PLCopen.org/OpcUa/IEC61131-3/Profile/Server/ControllerOperation |
| Controller Engineering Server Facet | http://PLCopen.org/OpcUa/IEC61131-3/Profile/Server/ControllerEngineering |
| Controller Engineering Client Facet | http://PLCopen.org/OpcUa/IEC61131-3/Profile/Client/ControllerEngineering |

## A.3    Namespace for IEC61131-3 Function Blocks

The namespace for all *Ctrl Function Block Type* nodes defined in other PLCopen documents like *Motion Ctrl Function Blocks* is "http://PLCopen.org/OpcUa/IEC61131-3/FB/".

The CSV file containing the numeric identifiers for this namespace can be found here:

http://www.PLCopen.org/OpcUa/IEC61131-3/FB/NodeIds.csv

The *NodeIds* for the defined nodes are composed of this namespace and the numeric identifier for the defined node.

## Annex B (informative): PLCopen XML Additional Data Schema

### B.1   XML Schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:pra="http://www.plcopen.org/xml/tc6_0200/OpcUa"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.plcopen.org/xml/tc6_0200/OpcUa">

  <xsd:simpleType name="AccessLevel">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Read" />
      <xsd:enumeration value="Write" />
      <xsd:enumeration value="ReadWrite" />
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="Visible">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Yes" />
      <xsd:enumeration value="No" />
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="InstrumentRange">
    <xsd:attribute name="Low" type="xsd:double" use="required" />
    <xsd:attribute name="High" type="xsd:double" use="required" />
  </xsd:complexType>

  <xsd:complexType name="EuRange">
    <xsd:attribute name="Low" type="xsd:double" use="required" />
    <xsd:attribute name="High" type="xsd:double" use="required" />
  </xsd:complexType>

  <xsd:complexType name="EUInformation">
    <xsd:sequence>
      <xsd:element name="DisplayName" type="LocalizedText" minOccurs="1" maxOccurs="1"/>
      <xsd:element name="Description" type="LocalizedText" minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="NamespaceUri" type="xsd:string" use="optional" />
    <xsd:attribute name="UnitId" type="xsd:int" use="optional" />
  </xsd:complexType>

  <xsd:complexType name="InstanceInformation">
    <xsd:sequence>
      <xsd:element name="NodeId" type="xsd:string" minOccurs="1" maxOccurs="1"/>
      <xsd:element name="InstanceNamespaceUri" type="xsd:string" minOccurs="1" maxOccurs="1"/>
      <xsd:element name="Delimiter" type="xsd:string" minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="LocalizedText">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="Key" type="xsd:string" use="optional" default="" />
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="NodeId">
    <xsd:sequence>
      <xsd:element name="Identifier" type="xsd:string" minOccurs="0"
       maxOccurs="1" nillable="true" />
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```