DRIVECOM

DriveServer

Version 1.1, 19.October.01 09:38

Editor: DRIVECOM Nutzergruppe e.V.

Postfach 1102, D-32817 Blomberg

Phone : ++49 52 35 / 3-4 18 64

Fax : ++49 52 35 / 3-4 18 62

Internet : http://www.DRIVECOM.org

Authors:

Albach	Lust	Lahnau
Arlt	Lenze	Hameln
Hadlich	Ifak system GmbH	Magdeburg
Iwanitz	Softing GmbH	München
Krumsiek	Phoenix Contact	Blomberg
Leurs	Rexroth Indramat	Lohr am Main
Mirbach	Lenze	Hameln
Müller	Phoenix Contact	Blomberg
Pollmeier	ESR Pollmeier	Ober-Ramstadt
Riedl	Ifak Magdeburg e.V.	Magdeburg
Schleicher	Indramat-Refu	Metzingen
Schnurbusch	Lenze	Hameln
Ziegler	SEW-EURODRIVE	Bruchsal

For any remarks or comments please contact Stefan Pollmeier under gl@esr-pollmeier.de.

History of the document:

Version	Name	Company	Comment
0.0.22	Riedl	ifak	Creation
0.1	Mirbach	Lenze	- Three-stage addressing (BusServer) - Identification of the bus system by the DriveServer - Different networks or strings connected to a BusServer
1.0	Mirbach	Lenze	- Clarification of the bit-by-bit access to the DriveServer interface for the OPC client - Deletion of the bit-by-bit access to the BusServer interface
1.0a	Mirbach	Lenze	Type VT_ARRAY added to chapter 5
1.1	Mirbach	Lenze	Profidrive parameter channel information added to chapters 3.2.1.1 and 3.3.2.1. Do not indicate a subindex for parameters without a subindex. Due to the compatibility with the Profibus, the version 1.0 is now not compatible with version 1.1

Contents

1 Introduction

1.1 Overview

Intelligent modular systems contain mechanical, electronic and software components as well as sensors and actors which determine the production strategy. Modern controllers, for instance, take over extensive technological control and regulation tasks. Standardised fieldbus systems, such as Profibus, help to implement these modular machinery concepts.

Problems. From the user`s point of view a drive is something completely different than it is from the communication`s point of view. Users work with parameters, which are detectable by names. For communication, parameters are defined by number pairs (e.g. index, subindex; slot, index). Many users accept and use this way of representation and they know that the communication usually depends on the communication protocol or profile, i.e. handling differences occur. This results in a considerable engineering effort. A quick and controller-oriented data access would make engineering a lot easier and therefore cheaper.

The DRIVECOM user group therefore decided to standardise the communication interface for accessing drives. The **DriveServer** specification is based on the OPC interface standard, version 2.0. The innovative concept standardises the presentation and access to controllers and functions.

Due to the use of the OPC interfaces in process automation the individual adaptation to controllers and hardware is not longer necessary, but vendors of automation devices can do that themselves. Since the access to process data has been standardised and the software of the PC can be loaded via OLEs, adaptations to vendor-specific interfaces are not longer required. Up to now this process was mainly limited to communication networks, but now the DriveServer specification uses this concept for automation devices in general.

Integration into the OPC architecture. The DriveServer is based on the OPC technology. It is comparable to a layer between a user program with OPC client interface and the communication media. The connection with the communication media can either be implemented by the vendor in the DriveServer or a communication OPC server.

The DriveServer encapsulates device features according to their functions. All features are accessed via a functional interface. Thus the functionality of the devices, their description and access mechanism remains unchanged. Access is made via standard means, for instance, not parameters and their content but the access to these parameters are defined by names. The main advantage is that the vendors do not have to change their implementations.

The DriveServer specification is based on a clear separation between communication functionality and DriveServer functionality. By this it is ensured that the great variety of available and reasonable communication media can be used. The DriveServer communicates via an OPC interface. This open architecture ensures the flexibility of fieldbus systems required. This is another important step towards open and uniform automation solutions.

The OPC based integration of drives into engineering systems described here sets a new standard for fieldbus communication. Its mainly independent architecture opens up a wide field of applications for the DriveServer. If this concept will also be accepted for fieldbus components, it will be the basis for device servers in general.

This document describes a uniform application interface for accessing drives. The aim of this specification process is to create a plug & play device by representing drive variables and vendor-specific functions for OPC servers, like driver software for printers.

2 Architecture

2.1 Overview

The DriveServer architecture is completely based on the OPC specification and thus ensures compatibility with other servers. The DriveServer specifications can be implemented by OPC servers, but they can also be met with standard OPC means. At first the DriveServer itself is a client of a communication OPC server (called BusServer in the following text) since the communication interfaces usually connect fieldbuses. The BusServer can however be a server of any transfer medium.

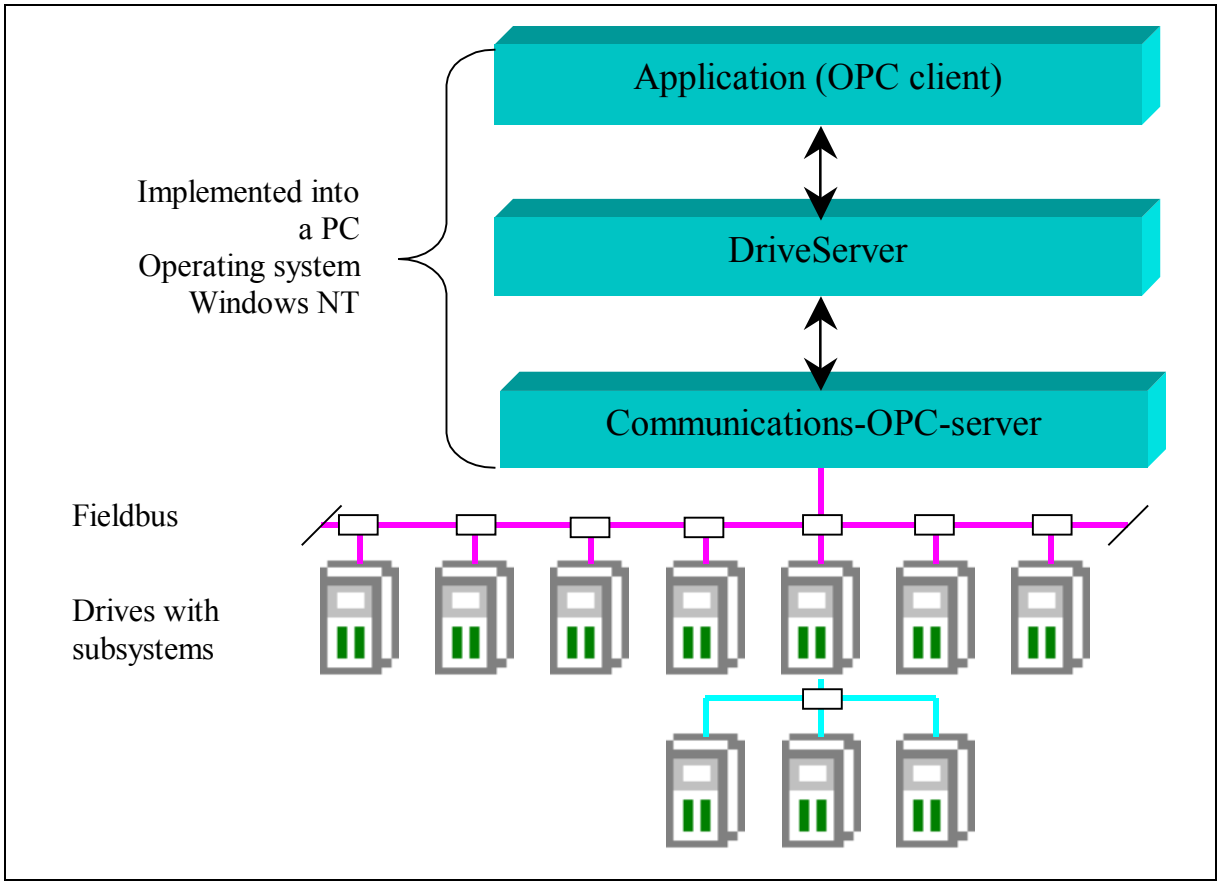

Figure 1: OPC server architecture

2.2 Layout of the components

The software components shown in figure 1 can be installed on individual computers or computers connected via a network. For this, the servers must meet the OPC and (D) COM specifications stipulated for the registration on a computer or computer network. These specifications do not describe the installation since it is defined by the DCOM standard mechanisms.

A common application is the access of a DriveServer to one or several BusServers. The configuration determines which BusServers will be accessed by the DriveServer. The DriveServer vendor defines how the configuration is to be made.

The DriveServer must log in at the BusServer with `IOPCCommon::SetClientName` (e.g. „`DriveServer_XY.EXE`“ or „`\\192.168.10.2\DriveServer_XY.EXE`“) according to the OPC specification. If necessary, the BusServer can thus detect the client.

Dots are defined as OPC delimiters („.“). Therefore dots cannot be used as part of a name for a parameter.

2.3 Communication structure

The DriveServer maps the device-related accesses of the application to fieldbus-related accesses. According to the OPC specification, DriveServer and communications OPC server access use names for accessing. These names can be preconfigured or have a dynamic character, depending on the server implementation. Figure 2 shows the communication using names and the mapping of bus and device architecture.

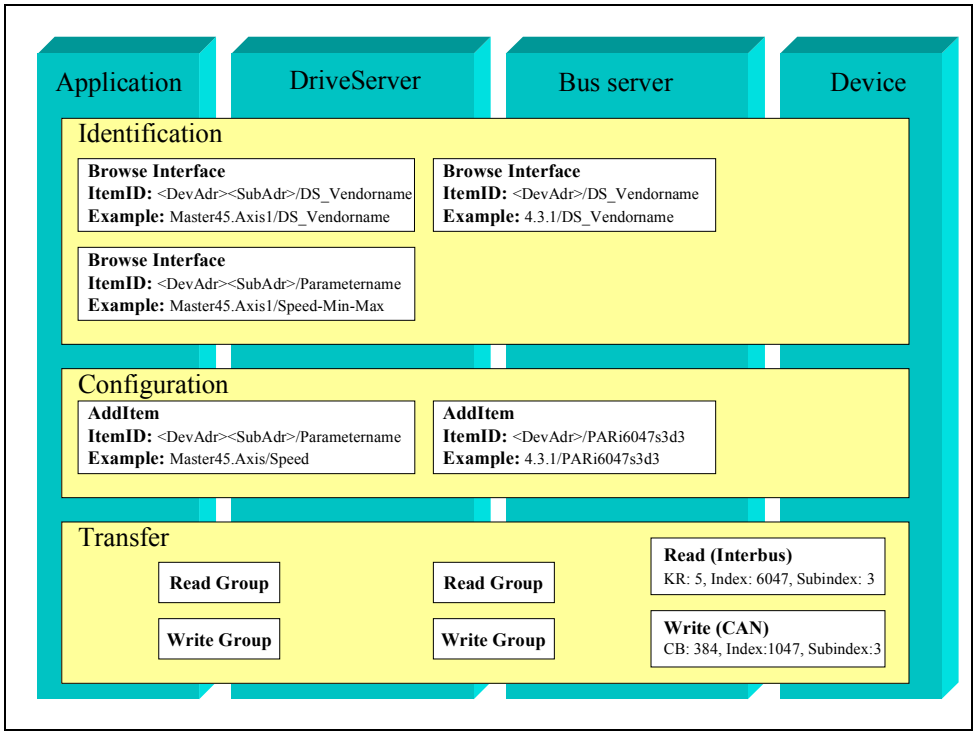

Figure 2 : Communication structure

The interface between application, DriveServer and BusServer comprises three phases:

- Device identification

  The DriveServer detects devices available at the bus via the browse interface of the BusServer. This detection follows the regulations described under 3.3.2. The device types are determined via a vendor-specific identification algorithm which is implemented in the DriveServer and which also allows the device types to be displayed in the browse interface of the DriveServer. An application can now select the devices.

- Item definition as access specification (configuration)

  As soon as it is clear which devices are to work together, OPC groups can be created in the DriveServer. These OPC groups contain OPC items which correspond to the controller parameters. Based on the previous controller identification the DriveServer can now create its own name space and reject parameter queries, if necessary. Thus faulty queries can be avoided.

In general, the OPC groups and OPC items created in the DriveServer are also created in the corresponding BusServers. Here the OPC item names correspond to the bus and controller-internal addresses. If an OPC group or OPC item cannot be created in the BusServer, also the DriveServer must indicate that these objects cannot be created.

- I/O operations (transfer)

  After the server configuration, the data transfer can be activated by the application, which means that read and write commands are executed by the groups of the DriveServer so that all OPC items included in the OPC group communicate. These commands are passed on to the subordinate BusServers which then exchange parameter values with the controllers.

2.4 DriveServer architecture

According to the OPC definition a DriveServer provides an OPC server interface and an OPC client interface. The DriveServer consists of three main components:

- A component which implements the OPC server interface
- A component which implement the DRIVECOM features and thus is available for the connection of all controllers but however allows different implementations
- A component which groups the vendor-specific features of the devices

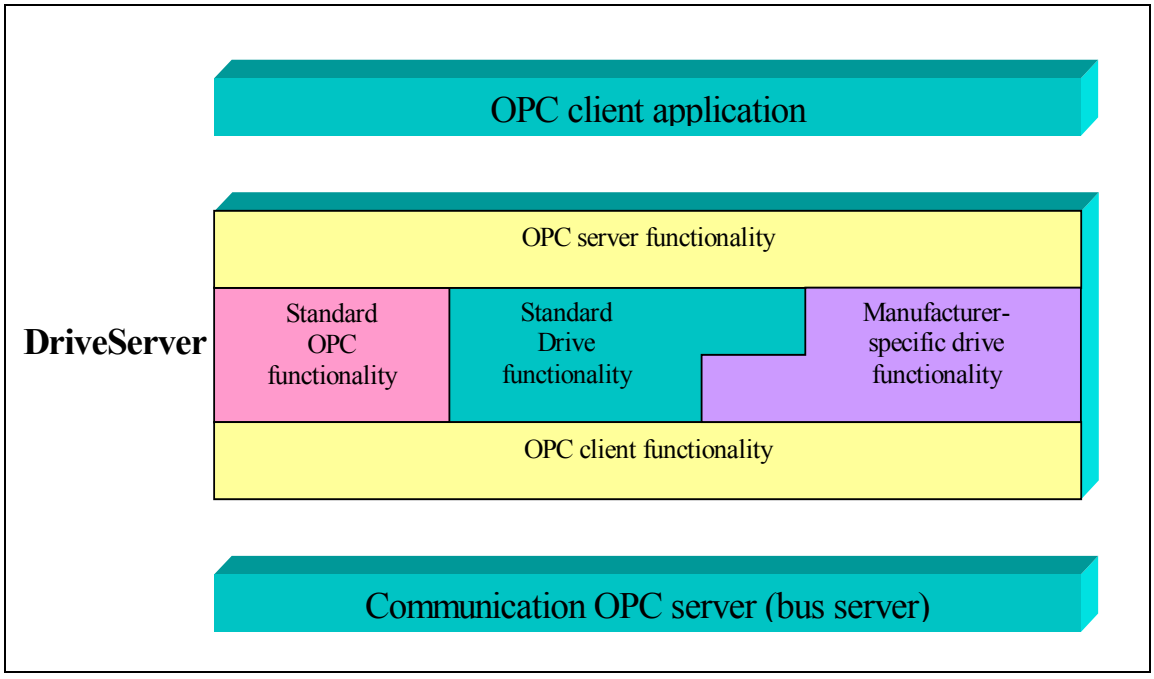

Figure 3: DriveServer architecture

2.5 BusServer architecture

With this architecture the BusServer communicates with the controllers. In general the BusServer is a bus-specific OPC server, i.e. read and write requests cannot be determined with this specification.

The BusServer does not have to directly access a bus, it is for instance possible that an OPC server is just based on TCP/IP and only uses certain sockets of the operating system or communicates with other (D)COM components which simulate controllers.

The requirements to BusServers are limited to the standard OPC server functionality. The browse interface must however be implemented. Dynamic configuration of the BusServer should be possible, i.e. OPC items can be added during run time although they have not been configured in the first place. All requirements listed under 3.3 must be met.

3 Functionality

3.1 Overview

The OPC specification “Data Access 2.0” defines the structure and the response of interfaces. Everything related to the name space of a server (structure, itemID, etc.) is not defined which means a certain freedom for the developing engineer and uncertainty for the user.

The DriveServer specification limits this freedom in creating the name space and the semantics of itemIDs. As a result, automated processes and easy handling (recognition) can be ensured for the user.

3.2 DriveServer functionality

A DriveServer provides the functionality necessary to access and handle a drive.

3.2.1 Overview

The controllers can be automatically identified via the browse interface of the DriveServer. Exactly defined <keyword_tag> are assigned to every controller. This specification proposes the ‘DS_Vendorname‘ as <keyword_tags>, it is however possible to define different strings. The DriveServer client accesses the name space of the DriveServer, which has a flat format according to the OPC specification (see page 9) and searches the name space for possible controller addresses following the identifying <keyword_tag>. According to its definition the complete text of the filtered name, without the DS_Vendorname determines the controller address, it is called <device_tag>. The syntax of this text depends on the server. Since this text indicates a clearly defined access path, it is used by the clients instead of a controller address. Thus the client does not need the bus-specific controller address to access the controllers.

In an OPC name space the DriveServer indicates the drives detected. This name space has a hierarchical structure, i.e. the name space has a structure (branch/leaf) which is similar to the file structure in a computer (directories/files). A branch corresponds to a directory and a leaf to a file (OPC server data).

The name space can be queried via the IOPCBrowseServerAddressSpace interface. The client can browse through the different structure levels in the name space (similar to a change of directories) and query leafs of individually structured items. The name space can also be queried using flat format (completely or partly). The leaf names are indicated together with the names of the higher-level structure items (similar to absolute file names).

Example: 2 controllers are connected to the bus. They are assigned to different addresses.
Hierarchical name space:

```
Bus Server Name
      Address 1
            DS_Vendorname
            DS_Devicename
            DS_DeviceID
            ...
      Address 2
            DS_Vendorname
            DS_Devicename
            DS_DeviceID
```

```
...
```

flat format

```
Bus Server Name.Address 1.DS_Vendorname
Bus Server Name.Address 1.DS_Devicename
Bus Server Name.Address 1.DS_DeviceID
...
Bus Server Name.Address 2.DS_Vendorname
Bus Server Name.Address 2.DS_Devicename
Bus Server Name.Address 2.DS_DeviceID
...
```

If the DriveServer addresses several BusServers, a corresponding structure level can be created in the name space of the DriveServer. Every structure item of this level corresponds to a BusServer. The drives, which communicate with the BusServers, can be assigned to the corresponding BusServer structure items. The string for the BusServers displayed while browsing is freely selectable and can be configured (depending on the implementation). The name of the hierarchy level for the controller address is created by concatenating the value of the OPC item `DS_BusPort`, „-“ and the value of the OPC items `DS_DeviceID` of the BusServer.

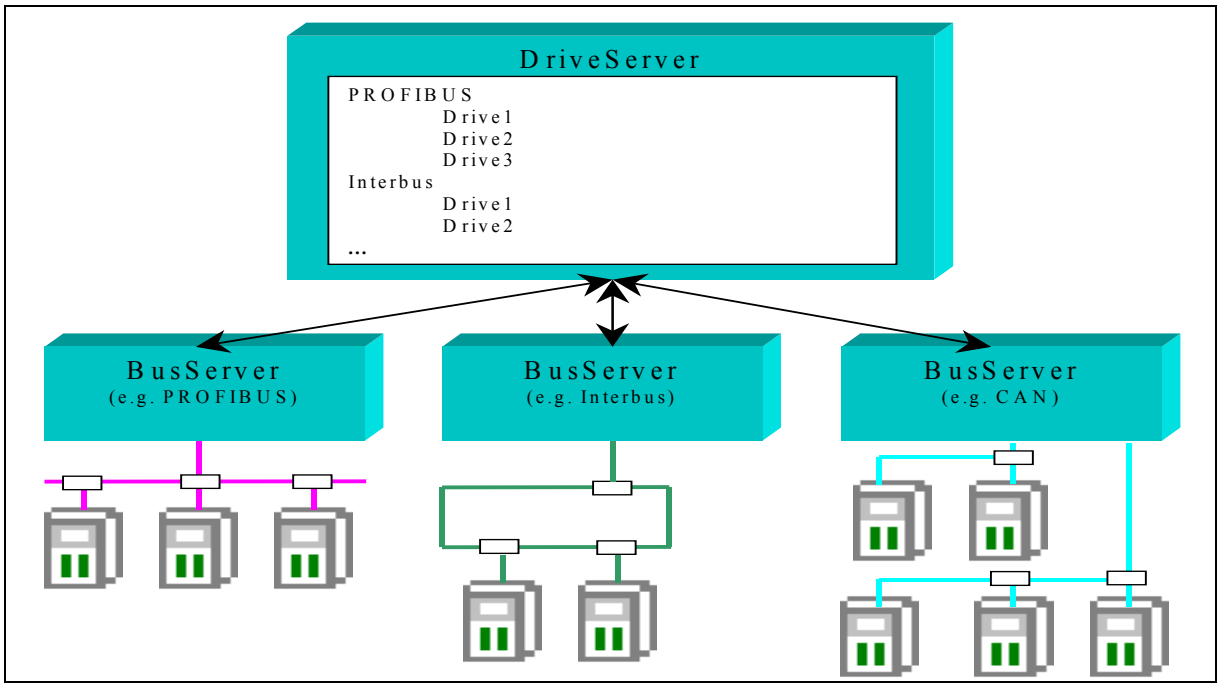

Figure 4: Drive representation in the DriveServer

The name space hierarchy shows parts of the system structure. Slave drives have a lower position in the hierarchy than the corresponding master drives.

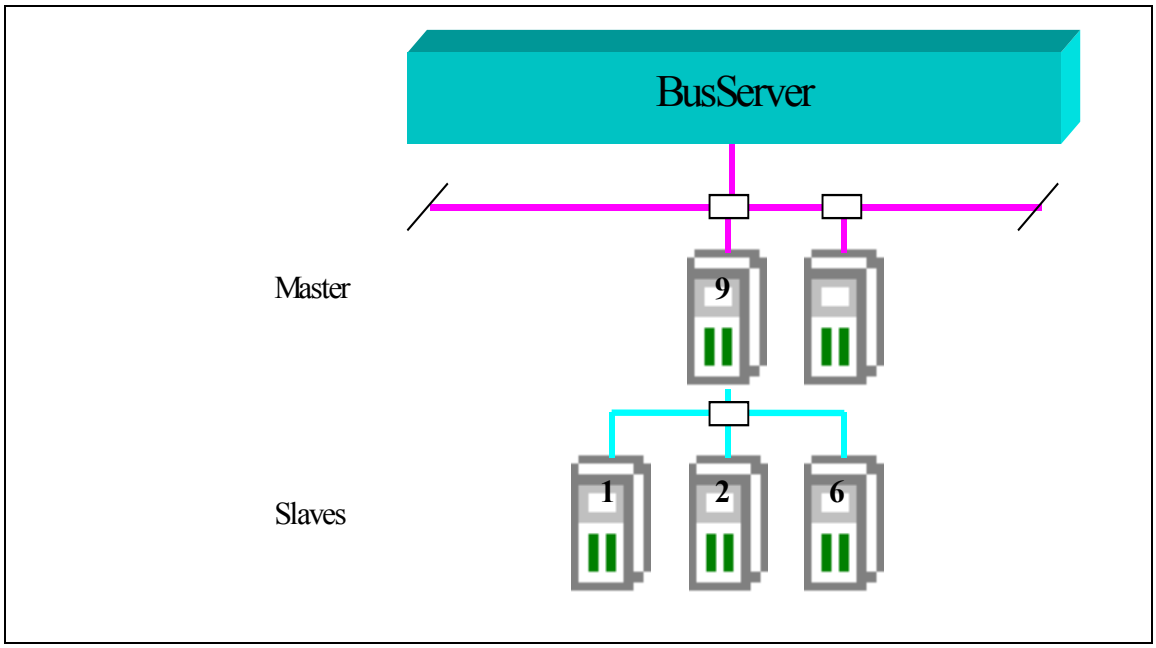

Figure 5: Drive with subsystems

Example for the name space of a drive with subsystems:

```
BusServer name
      9
                  DS_Vendorname
                  DS_Devicename
                  DS_DeviceID
                  ...
                  1
                        DS_Vendorname
                        DS_Devicename
                        DS_DeviceID
                        ...
                  2
                        DS_Vendorname
                        DS_Devicename
                        DS_DeviceID
                        ...
```

The initialisation of the name space depends on the implementation (which is vendor-specific) and is also called server configuration. We distinguish between two configuration types:

- Static configuration: Static configuration means either reading in the configuration or the controller description when starting the server. According to the DriveServer specification and also the vendor's specification any symbolic name can be used.

- Dynamic configuration: Dynamic configuration is made through ItemIDs with a defined syntax.

Every controller can be clearly identified in the name space by means of a <keyword_tag>. All parameters of a controller are defined as leafs of a controller-specific branch. Therefore all parameters are structured as followed in a name space with a flat format:

```
<item_tag>          := <device_tag><parameter_tag>
```

The <device_tag> is assigned by the DriveServer itself and contains all characters and strings to identify a controller even if controller-specific character strings and <parameter_tags> are separated by a delimiter.

If it is necessary, the string <device_tag> can be directly in front of a parameter address to find a controller parameter.

3.2.1.1 Reserved names (DriveServer-specific names)

This specification reserves specific names (observe small and capital letters) for certain standard functions.

Every controller provides standard parameters which can be used by the client to identify the controller. The following parameters are of data type VARIANT. The data type can be freely selected by the vendor since every OPC client can interpret or convert the data type as required.

Parameter	Meaning	m/o
DS_Devicename	Controller name	M
DS_DeviceID	Controller identification (bus address, URL, ...)	M
DS_Vendorname	Vendor's name	M
DS_Devicetype	Controller type	O
DS_Vendorcode	Vendor's code	O
DS_ProfileID	Profile identification	O
DS_BusSystem	Component category (registry format) for a fieldbus system (see chapter 4)	O
DS_BusPort	Bus network or string (see chapter 3.3.2.1)	O

Examples :

```
Bus Server Name.9.DS_ProfileID (DS_DeviceID = 9)
Bus Server Name.127-168-0-2.DS_ProfileID (DS_DeviceID = 127-168-0-2)
Bus Server Name.COM1-9.DS_ProfileID (DS_BusPort = COM1, DS_DeviceID = 9)
```

Name convention

Individual drive parameters can be addressed using a defined syntax. The following name convention applies to all parameters (also those with a symbolic name):

The optional bit-by-bit access to process and parameter data can be limited to certain parameters by the DriveServer.

```
<parameter_tag>      := <parameter_channel> | <process_channel>
<parameter_channel>  := "PAR"
                        <index_info><subindex_info>[<datatype_info>]
                        [<bit_info>][<length_info>]
                        [<extra_info>]
<process_channel>    := <process_cannel_info>
                        <index_info><subindex_info>[<bit_info>]
                        [<datatype_info><length_info>[<extra_info>]

<process_channel_info> :=    "OUT" | "IN"
<index_info>         := (I|i)[0-9]+
<subindex_info>      := (S|s)[0-9]+
<bit_info>           := (B|b)[0-9]+
<datatype_info>      := (D|d)(VT_UI1 | VT_ARRAY | ... )
                        (The numbers to be used here are listed in chapter 5)
<length_info>        := (L|l)[0-9]+
<extra_info>         := (X|x)<String>
```

`index_info:`	Object index (starts with 0)
`subindex_info:`	For process data channels access to a single byte if the objects consists of several bytes, 0 means access to all bytes of the object. In parameter data channels, the subindex indicates the position in indexed fields. If the addressed object is not a subindexed parameter, a subindex must not be indicated.
`bit_info:`	Bit position in the object, counting starts at LSB with bit number 0; if the bit position is indicated, also the length must be defined
`length_info:`	Length of the data to be transferred in bits

Examples :

`PARI1000S0:`	Subindexed parameter variable with index 1000 and subindex 0. The DriveServer already knows the data type from the device description. Therefore it is not necessary to indicate the data type.
`pari120d3 :`	Non-subindexed parameter variable with index 120, data type 3 (corresponds to `VT_I4`: 4-byte integer)

`pari10s40d8 :`	Parameter variable with index 10 and subindex 40, data type 8 (corresponds to `VT_BSTR`: character string)
`INI2S0D3:`	IN process variable on process data channel 2. The entire process variable is to be interpreted as 4-byte integer `(VT_I4)`.
`INI2S0D3L16:`	IN process variable on process data channel 2. The last 16 bits of the variables are to be mapped onto a 4-byte integer variable. Assumption: Width of the process data channel is 4 bytes. Source: `MSB XXXXXXXX XXXXXXXX PONMLKJI HGFEDCBA LSB` Target: `MSB XXXXXXXX XXXXXXXX PONMLKJI HGFEDCBA LSB`
`INI1S2D3B2L4 :`	IN process variable on process data channel 1. Byte 2 is to be mapped onto a 4-byte integer variable `(VT_I4)` starting from bit 2 over 4 bits. Assumption: Width of the process data channel is 4 bytes. Source: `MSB XXXXXXXX XXXXXXXX XXDCBAXX XXXXXXXX LSB` Target: `MSB XXXXXXXX XXXXXXXX XXXXXXXX XXXXDCBA LSB`
`INI1S0D3B10L4 :`	IN process variable on process data channel 1. The bits 10, 11, 12, 13 of the process data are to be mapped onto a 4-byte integer variable (`VT_I4`). Assumption: Width of the process data channel is 4 bytes. Source: `MSB XXXXXXXX XXXXXXXX XXDCBAXX XXXXXXXX LSB` Target: `MSB XXXXXXXX XXXXXXXX XXXXXXXX XXXXDCBA LSB`
`PARI150S0D3B1L4:`	Parameter variable with index 150 and subindex 0. 4 bits are to be mapped onto a 4-byte integer variable (`VT_I4`) starting from bit 1. Source: `MSB - XXXXXXXX XXXDCBAX - LSB` Target: `MSB XXXXXXXX XXXXXXXX XXXXXXXX XXXXDCBA LSB`

With complete path:

```
Bus Servername.9.pari10s40d3
Bus Servername.9.ini1s0d3l4
```

By means of this syntax the parameter access can be mapped onto symbolic names in the DriveServer (or vice versa).

3.2.1.2 Structure of the name space

This specification is based on a hierarchical name space.

Multi-axis drives

The DriveServer encapsulates the controller architecture. One of the controller features is to create subsystems which are called multi-axis drives.

Drives can be connected by means of different communication systems. These systems are not necessarily standardised but always selected by the vendor. The drives communicate via vendor-specific protocols which need to be encapsulated.

The DriveServer provides a system architecture for other applications which ensures direct access of all controllers of the subsystem. Access to the architecture is possible because of the name space which can be created in the DriveServer. While reading and writing items, the DriveServer maps the data to the controller protocol.

All controllers identified via the browse interface of the BusServer are tested when the DriveServer creates its internal name space. It identifies master and slave drives via a vendor-specific protocol.

Based on these information the DriveServer creates a subhierarchy for every controller. The controllers can be identified by the application. The subsystem is also marked with a <keyword_tag>.

3.2.2 Parameter set transfer

Every drive has a subhierarchy for the parameter set transfer:

Parameter	Meaning	m/o
DS_ParameterSets	Number of parameter sets in the drive	O
DS_ParameterSet[0..9]+	Subhierarchy for each parameter set for transfer actions	O
DS_ParameterSetAll	Subhierarchy transfer actions of all parameter sets	O

```
DS_ParameterSet[0-9]+
            Filename
            Action
            State
            Result
DS_ParameterSetAll
            Filename
            Action
            State
            Result
```

Every parameter set has got this subhierarchy. The individual parameter sets are unambiguously identified by numbers.

- Use the item 'Filename' to enter a file name. The file contains the vendor-specific transfer data. The file format is also vendor-specific. Filename is optional. If the user does not enter a Filename, the current data of the DriveServer are written to the drive or overwritten from the drive.

- The item 'Action' describes the transfer type. The following strings can be written to the item:

  - "UPLOAD"
  - "DOWNLOAD"
  - "VERIFY"
  - "COMPARE"

  Written to the item starts the respective procedure. The results of VERIFY and COMPARE are stored in files. The file names are created from the item value for 'Filename' and a corresponding extension. The format is selected by the vendor.

- The item ‘State‘ can only be read. It informs about the current state of the transfer operation. The following states are defined:
  - ”READY”
  - ”RUNNING”
  - ”UNDEFINED”

  Writing to the ‘Action‘ item only starts an action if the ‘State‘ is ”READY”.

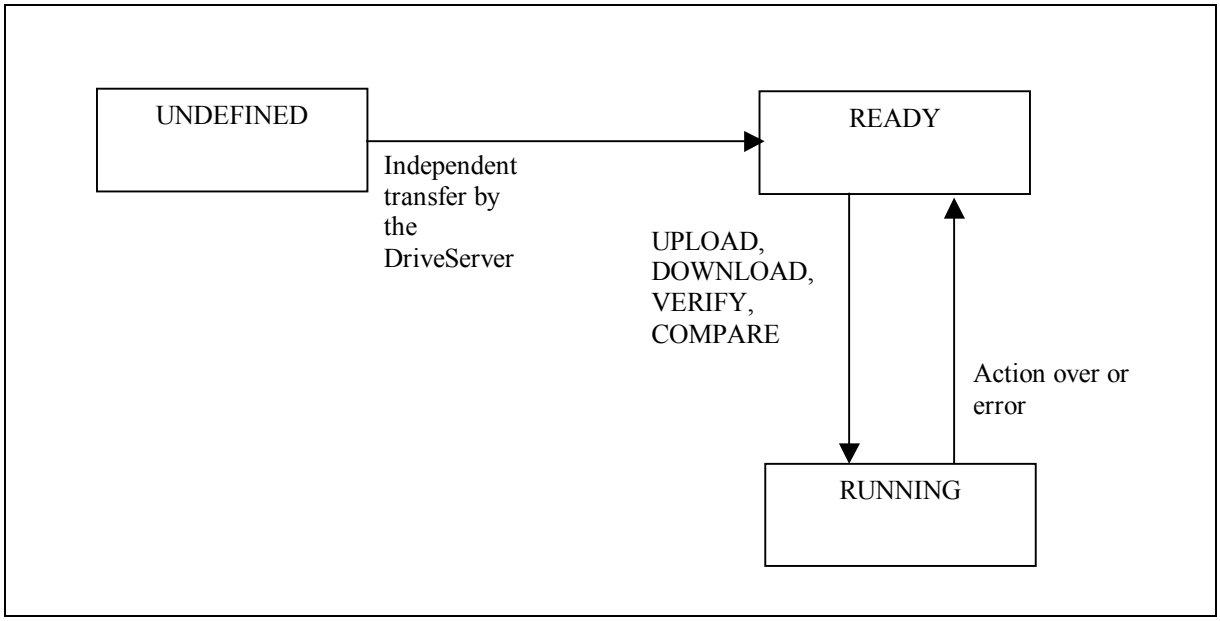

Figure 6: State transition for parameter set transfer

- The ‘Result‘ item can only be read. It informs about the success of an action. This item should only be evaluated when ‘State‘ is ”READY”. ‘Result‘ has the value range and the coding of HRESULT.

3.2.3 User programs for the drive

Every drive has a subhierarchy for user programs:

Parameter	Meaning	m/o
DS_Programs	Subhierarchy for drive programs	O

```
DS_Programs
        ProgramName1
                Filename
                Action
                State
                Result
                CodeAttribute
        ProgramName2
```

```
                    Filename
                    ...
                ...
```

Every drive program has its own subhierarchy. The individual programs are identified by unambiguous and freely selectable node names (here: `ProgramName1`, `ProgramName2`).

- The item ‘`Filename`‘ describes a file name. The file contains vendor-specific program data. The file format is also selected by the vendor. `Filename` is optional.
- The item ‘`Action`‘ indicates the transfer type. The following strings can be written to the item:
  - ”`UPLOAD`”
  - ”`DOWNLOAD`”
  - ”`PREPARE`”
  - ”`START`”
  - ”`STOP`”
  - ”`HALT`”

  Writing to an item starts the respective procedure. The values can be written depending on the status (see Figure 7).
- The item ‘`State`‘ can only be read. It informs about the current status of the transfer operation. The following states are defined:
  - ”`INITIALIZED`”

    The drive has not been loaded with a program code. This is the default status.
  - ”`LOADED`”

    A program code has been loaded to the drive. The program can be prepared for being started by ”`PREPARE`”.
  - ”`RUNNING`”

    The program is running.
  - ”`STOPPED`”

    The program has been stopped. It can be restarted any time and will start at its beginning.
  - ”`HALTED`”

    The program has been stopped. A restart is possible any time. It will restart where it had been interrupted.

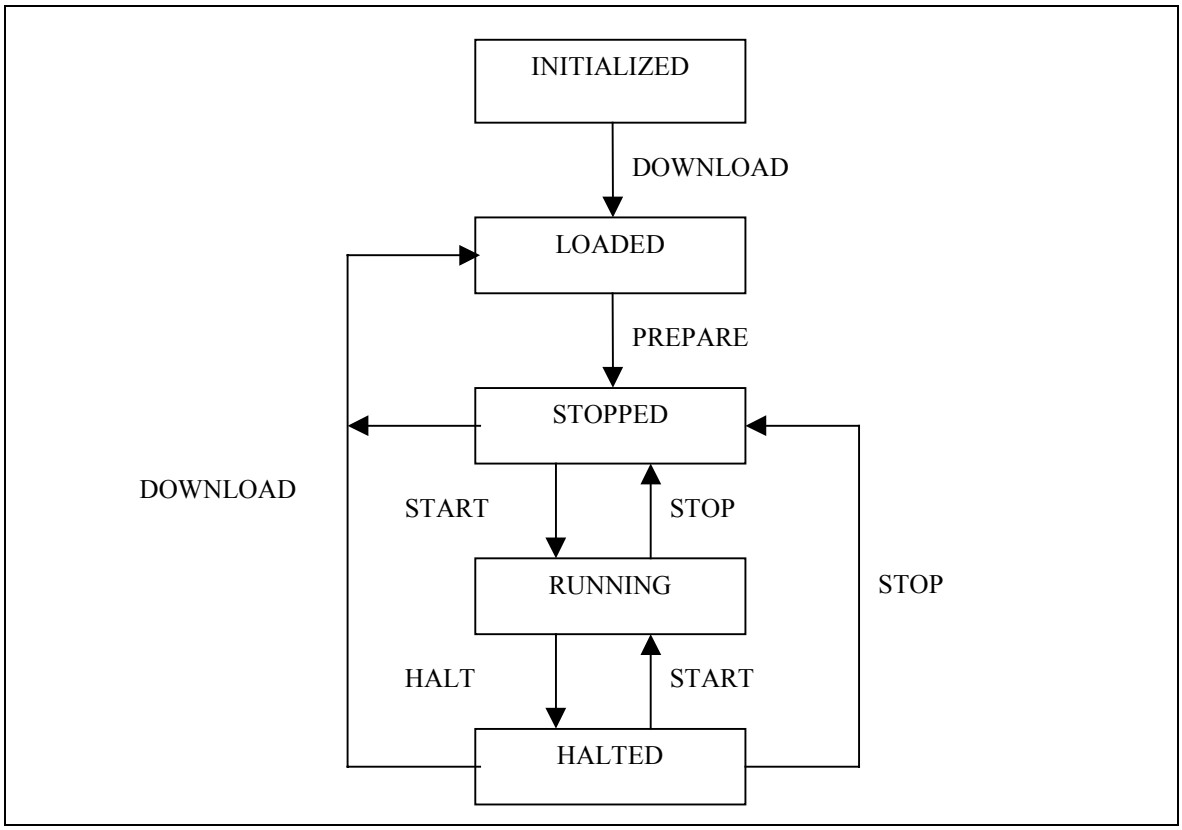

Figure 7: State transitions of a program

- The item ‘`Result`‘ can only be read. It informs about the success of the completed action. ‘`Result`‘ has the value range and the coding of `HRESULT`.

- The item ‘`CodeAttribute`‘ defines the start type of the program after a restart of the drive (value range: `VT_I4`):

Value	Meaning
0	The program does not start on its own after a restart of the drive.
1	The program starts on its own after a restart.
2 ... EFFF	Reserved
FFFF ... -1	Vendor-specific

The program transfer and operation are individually selected by the vendor.

3.2.4 Drive identification

The DriveServer identifies the drives available in the BusServer by means of the following algorithm:

1. Browsing of the name space in the BusServer and searching for the `<keyword_tag>` `DS_Vendorname`. The result should be formatted in `flat`.

2. The Item Ids for the found leafs are to be queried and dissected into controller and parameter specific components.

3. The DriveServer creates the items required for the drive identification in the BusServer and evaluates them according to the vendor's instructions.

4. After the evaluation, these items can be released for the BusServer.

The DriveServer is now responsible for presenting the controllers it detected as operator-accessible to its clients.

3.3 BusServer functionality

3.3.1 Overview

A BusServer enables communication between system values using a certain communication medium. BusServers are vendor specific and are not mainly described in this specification. The following requirements must be met to ensure optimum co-operation between DriveServer and BusServer:

- A BusServer must support the optional OPC interface ‚`IOPCBrowseServerAddressSpace`'.

- Name spaces should be extendable by, for instance, creating an OPC item for which no item is defined in the name space. It is thus possible to create OPC items which have not been introduced to the name space of the BusServer by static configuration earlier.

- The BusServer must register itself with the corresponding component categories for the supported bus systems.

3.3.2 Name space

The name space of the BusServer is created and structured by the vendor according to communication aspects.

The following rules must be met to ensure co-operation between DriveServer and BusServer:

- The reserved names of the BusServer must have been created earlier and the spelling must not be changed.
- It must be possible to search the name space by the DriveServer in `flat` format.

The name space is initialised by the vendor. There are two types of configuration possible:

- Static configuration: Static configuration means either reading in the configuration or the controller description when starting the server. According to the DriveServer specification and also the vendor's specification any symbolic name can be used.
- Dynamic configuration: Dynamic configuration is made through ItemIDs with a defined syntax.

Every controller is to be unambiguously identified by means of a <`keyword_tag`>.

All parameters of a device can be found as leafs of a device-specific branch.

3.3.2.1 Reserved names

This specification reserves names for certain standard functions.

BusServer-specific names

Every device has standard parameters which help the client (DriveServer) to identify a device (capital and small letters).

Parameter	Meaning	m/o
`DS_Devicename`	Controller name	M
`DS_DeviceID`	Controller ID (bus address)	M
`DS_Vendorname`	Vendor's name	M
`DS_Devicetype`	Controller type	O
`DS_Vendorcode`	Vendor's code	O
`DS_BusSystem`	Component category (registry format) for a fieldbus system (for definition see chapter 4)	O
`DS_BusPort`	Bus network or string	O

Notes:

The parameter `DS_DeviceID` must be unambiguous for the corresponding bus string (BusServers can support several bus strings, e.g. several PC cards, several ports at a PC card), since it is used for the creation of the DriveServer name space. If a BusServer has several bus networks or strings, the `DS_DeviceID` is not unambiguous any more since, for example, address 5 can occur at every bus string. With several bus strings at the BusServer, the BusServer must mark bus string in the parameter `DS_BusPort` so that the name space structure is unambiguous (`DS_DeviceID` and `DS_BusPort`).

The sign "." must not be used for parameters since it is used as OPC delimiter.

The values of the parameters DS_Devicetype, DS_DeviceID, DS_BusSystem, DS_BusPort and DS_Vendorcode are independent of the language since they are used for addressing and identification.

If a BusServer supports several bus systems, the component categories registered do not indicate which bus the controller is connected to. In this case, the parameter DS_BusSystem gives all information necessary.

Name convention

The following name convention applies to parameters without symbolic names. Two and three stage addresses are possible (depending on the bus system).

Bus systems which use 2-stage addressing (e.g. CANopen) address parameters via an index and a subindex. This type of addressing corresponds to what the user usually sees. The 3-stage address is used for, for instance, DeviceNet applications. The <index_info> is an instance and the <subindex_info> an attribute. The class, which is the 3rd stage, requires an additional identifier. It can only be used for bus systems with 3-stage addressing.

```
<parameter_tag>      := <parameter_channel> | <process_channel>
<parameter_channel>  := "PAR"
                        [<class_info>]<index_info>[<subindex_info>]
                        <datatype_info>[<length_info>]
                        [<extra_info>]
<process_channel>    := <process_cannel_info>
                        <index_info><subindex_info>
                        <datatype_info><length_info>[<extra_info>]

<process_channel_info> :=     "OUT" | "IN"
<class_info>         := (C|c)[0-9]+
<index_info>         := (I|i)[0-9]+
<subindex_info>      := (S|s)[0-9]+
<datatype_info>      := (D|d)(VT_UI1 | VT_ARRAY | ... )
                        (The numerical values to be inserted are listed in chapter 5)
<length_info>        := (L|l)[0-9]+
<extra_info>         := (X|x)<String>
```

class_info:	Object class; only with 3-stage addressing
index_info:	2-stage addressing: Index of the object (starts with 0)
	3-stage addressing: Instance of a class
	Process data: Number of the process data channel
subindex_info:	2-stage addressing: Subindex of an object (starts with 0) If the addressed object is not a subindexed parameter, do not indicate a subindex.
	3-stage addressing: attribute of an instance

	Process data: Access to a byte if the object consists of several bytes. 0 means that all bytes of an object are accessed
`length_info:`	Length of the required in bits

Examples :

`INI1S0D3:`	IN process variable to process data channel 1. The object is to be completely mapped to a `VT_I4`.
`pari24564D3 :`	Parameter variable with index 24564, data type 3 (corresponds to `VT_I4`)
`pari24566S0D3 :`	Subindexed parameter variable with index 24566 and subindex 0, data type 3 (corresponds to `VT_I4`)
`parc100i10s40d8:`	Parameter variable class 100, instance 10, attribute 40 in a bus system for 3-stage addresses, representable as data type `VT_BSTR`

Complete path:

```
OPC://PROGID- Bus Server /DP://brd0.seg0.dev9/ini1s0d3l4
OPC://PROGID- Bus Server /DP://brd0.seg0.dev9/pari24566s0d3
```

3.3.2.2 Structure of the name space

This specification is based or a hierarchical structure of the name space. All parameters of a controller can be found as leafs of a controller-specific branch. Therefore all branch names in the name space (`flat` format) have the following structure:

```
<item_tag>          := <device_tag><parameter_tag>
```

The `<device_tag>` contains all signs and strings required by the OPC server for the identification of a controller including a possible delimiter between controller-specific character string and the `<parameter_tag>`.

4 Registration

DRIVECOM defines the following component categories.

All CATIDs and the corresponding descriptions are listed in the following.

“ DriveServer version 1.0”

```
CATID_DriveServer10 = {276BC1C0-0BC1-11D4-A78F-525405F5B2CF}
```

“ Busserver version 1.0”

```
CATID_BusServer10 = {276BC1C1-0BC1-11D4-A78F-525405F5B2CF}
```

“ **BusServer CANopen**”

```
CATID_CANOPEN = {3CF19FF1-907B-11d4-B4A7-0050DA3F121C}
```

“ **BusServer PROFIBUS-PA**”

```
CATID_PROFIBUS-PA = {3CF19FF2-907B-11d4-B4A7-0050DA3F121C}
```

“ **BusServer PROFIBUS-DP**”

```
CATID_PROFIBUS-DP = {3CF19FF3-907B-11d4-B4A7-0050DA3F121C}
```

“ **BusServer PROFIBUS-FMS**”

```
CATID_PROFIBUS-FMS = {3CF19FF4-907B-11d4-B4A7-0050DA3F121C}
```

“ **BusServer INTERBUS**”

```
CATID_INTERBUS = {3CF19FF5-907B-11d4-B4A7-0050DA3F121C}
```

“ **BusServer DeviceNet**”

```
CATID_DEVICENET = {3CF19FF6-907B-11d4-B4A7-0050DA3F121C}
```

“ **BusServer LON**”

```
CATID_LON = {3CF19FF7-907B-11d4-B4A7-0050DA3F121C}
```

It is demanded that the servers enter all categories supported by them into the registry (`HKEY_CLASSES_ROOT\Component Categories`) when they are installed. A description is stored together with the CATID value.

Every BusServer must enter all bus protocols supported as CATID. If a bus protocol is not identified by a CATID, the syntax defined under 3.3.2 is sufficient to address a certain parameter. Protocol conversions in the DriveServer are not necessary. The server itself declares its support of a CATID by corresponding entries under `Implemented Categories` in the registry.

5 Variant data types

The name convention uses the Microsoft variant data types. All possible types are listed in the following table.

Variant type	Value	Description
VT_EMPTY	0	Not standardised
VT_I2	2	2 byte integer with sign
VT_I4	3	4 byte integer with sign
VT_R4	4	4 byte floating point number
VT_R8	5	8 byte floating point number
VT_CY	6	Currency
VT_DATE	7	Date
VT_BSTR	8	String
VT_BOOL	11	Boolean, True=-1, False=0
VT_UI1	17	1 byte integer without sign

The data type VT_ARRAY contains data and information about the data type of array elements and array limits. The data type is indicated by a combination of the VT_ARRAY values and the variant type (table above) or by bit-by-bit OR. (Example: Value for the data type "Array of 1 byte integer without sign" 8209 (= 8192 or 17). Array length is variable).

Variant type	Value	Description
VT_ARRAY	8192	Array

6 Glossary

BusServer	OPC server which enables controller communication. The name space and the available server items depend on the communication
`<device_tag>`	Character string which represents the controller-specific part of an itemID
DriveServer	OPC server which enables controller (drives) communication. The name space and the available server items depend on the controllers.
`<item_Tag>`	Character string which represents a complete parameter address and can be used as itemID
`<keyword_tag>`	Keyword needed to find controllers in the name space and `<device_tags>` required for accessing parameters
Name space	Overview over the data known by the OPC server. A name space can have a hierarchical or a `flat` format, i.e. all data available is listed
`<parameter_tag>`	Character string which represent a parameter-specific part of an itemID

7 Literature

OPC OLE for Process Control, Data Access Custom Interface Standard, Version 2.0

www.opcfoundation.org