Modular EDSs and other EDS enhancements for DeviceNet

Viktor Schiffer, Rockwell Automation Germany

The current EDSs in use for DeviceNet are very versatile tools to configure slave devices, but they have certain shortcomings when modular devices are to be configured, specifically when it comes to determining the I/O sizes of the device to be configured and working with parameters in the modular section of the device. Using modular EDSs as already defined in the CIP common spec [1] in conjunction with a suitable software configuration tool opens up a whole scope of new capabilities. This paper describes several enhancements for EDSs including the concept of modular EDSs in a step by step approach, explaining the required and optional keywords in the set of EDSs required to describe a modular device and how these keywords interact. First, the concept of an assembly is introduced and it is shown how this assembly can be used in EDSs. In the next section, a set of minimum functionality modular EDSs (I/O only) is explained while the following section expands their functionality by adding configurable parameters in the modular parts of the device. Present and future capabilities of this concept are then discussed and how they can be applied to modular devices and even to devices that at first do not appear to be modular. Furthermore, the paper details the object and functionality enhancements required in a device to benefit from this concept. It also presents several sample EDSs and how they are used by the configuration tool.

INTRODUCTION

The use of EDSs for the configuration of DeviceNet devices has been around ever since DeviceNet has been made public. Recently, assemblies and modular constructs have been introduced that allow better configuration of slave devices. In particular, the extension into modularity helps overcoming the limitations of traditional EDSs. While the primary application of modular EDSs is expected to be for modular I/O (rack) systems, its use is not limited to that application. All configuration screen shots shown in this paper were taken with RSNetworx for DeviceNet 4.00.00 or higher.

ASSEMBLIES

The `[Assembly]` section describes the structure of data blocks. Often such a block is the data attribute of an Assembly object; however, this section of the EDS can be used to describe any complex structure. The description of this data block is similar to the mechanism that the Assembly object uses to describe its member list.

The contents of an assembly is described in Volume 2, chapter 4-3.5.8 of the DeviceNet specification [2]. All Assembly entries consist of fields that describe the general properties of the assembly and fields that describe the individual members of an assembly. While the overall size of an assembly is always described in bytes, all members of an assembly are described as a set of bits to allow all possible structures.

Figure 1 shows a sample entry in an `[Assembly]` section.

```
[Assembly]

Assem5 = " Configuration",      $ name
       "20 04 24 05 30 03",1,   $ path, size (in bytes)
       ,,,                      $ not used
       4, Param1,               $ # bits, reference
       3, Param2,               $ # bits, reference
       1, ;
...
```

Figure 1: Sample Assembly entry

This example shows how the Assembly #5 is split into one group of 4 bits, described by the `Param1` entry of this EDS, one group of 3 bits, described by the `Param2` entry of this EDS and one group of 1 bit without any further description.

USE OF ASSEMBLY ENTRIES IN AN EDS

In a normal (non-modular) EDS, Assemblies are mainly used in two cases: I/O Assemblies and Parameter Assemblies. The structure of the Assembly entry is identical in both cases: Bit groups are specified in size and described by `ParamN` entries.

```
[IO_Info]
...
Input1 = 1,              $ size = 1 byte
      4,                 $   4 significant bits
      0X000F,            $ compatible with all conections
      "1734 -IB4 Produce Connection",      $ name string
      6,                 $ path length
       "20 04 24 03 30 03",    $ assy obj, inst 3, attr 3
       "Producing connection contains state of the two inputs";
...
[Assembly]
Assem3 =    "1734 -IB4 Produce Assembly",
      "",                $ Path is implied (Assy obj attr3)
      1,                 $   total length in bytes
      0,,,               $ no changes, ",,"= reserved
      1, Param1,         $ # bits, Value for Input #0
      1, Param2,         $ # bits, Value for Input #1
      1, Param3,         $ # bits, Value for Input #2
      1, Param4,          $ # bits, Value for Input #3
      4,;                $ Pad bits
```

Figure 2: EDS showing an I/O Assembly

Figure 2 shows an example of how an assembly is used for an Input entry and Figure 3 shows how this is displayed in a configuration tool (slave side, similar display on the master side).

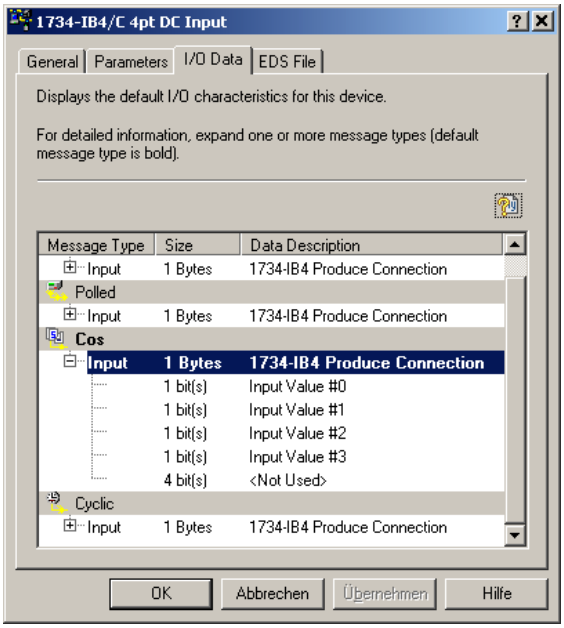

Figure 3: Use of an assembly in an Input entry

The entities described by `Param1` through `Param4` may be addressable attributes, but they do not have to. If a `ParamN` entry is used only to describe the components of an Assembly (without a path to an attribute), then it may be desirable to set the descriptor of this parameter to “non-displayed parameter”, [6].

Assemblies used as parameter assemblies allow the use of only one data structure containing all parameters of a device, but still maintaining access to the individual parameters. What differs from “normal” parameters is that up/download will always take place as an entity (the parameter assembly data is only one attribute!) and that an individual parameter may consist of any number of bits allowing packing of parameters, e.g. two nibbles in one byte.

```
[ParamClass]
    MaxInst=3;
    Descriptor=0x00;
    CfgAssembly=5;

[Params]

Param1 = 0,
      ,,                                  $ no path!
      0,0xC6,1,              $ descriptor, data type, data size
      "Bits 1 through 4","","",           $ name, unit, help string
      0,31,1,                             $ min, max, default
      ,,,,,,,,;                           $ other fields not used

Param2 = 0,
      ,,                                  $ no path!
      0x10,0xC6,1,           $ descriptor, data type, data size
      "Bits 5 through 7","","",           $ name, unit, help string
      0,3,2                               $ min, max, default
      ,,,,,,,,,;                          $ other fields not used
...
[Assembly]
      Assem5 =
      "Configuration","20 04 24 05 30 03",    $ name, path
      1,,,,        $ 1 byte, default descriptor
      5,Param1,    $ 5 bits described in Param1
      3,Param2;    $ 3 bits described in Param2
```

Figure 4: EDS with Parameter Assembly

Figure 4 shows an example of a Parameter Assembly and Figure 5 shows how this is displayed in a configuration tool.

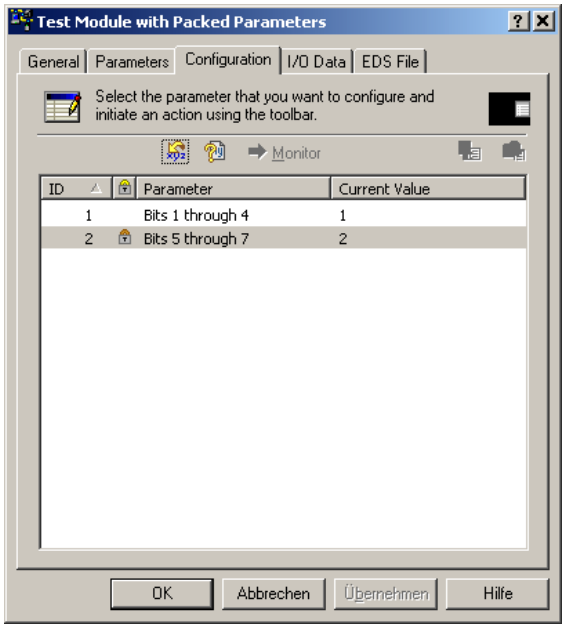

Figure 5: Use of a Parameter Assembly

GENERAL PRINCIPLE OF MODULAR EDSs

To make this explanation easier to understand an example of a modular I/O rack is used. Any modular device is made up of at least 3 components:

- A Chassis. This may physically be present or not. As a minimum, this is a logical “container” in which the other modules reside. A chassis has a maximum number of “slots” that may be populated.
- A module that connects to DeviceNet, a Communications Adapter. This module may or may not contain I/O data and/or configurable parameters.
- One or more I/O Modules that “populate” the Chassis. These connect to DeviceNet through the “Adapter”. Typically, these modules contain I/O data and/or configurable parameters.

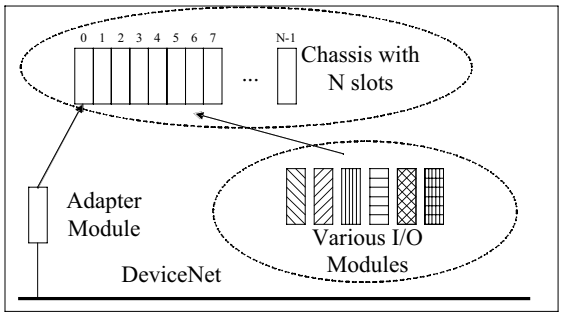

Figure 6: Relationship between the three Types of EDSs

Any system described by modular EDSs is put together with a minimum set of three EDSs:

1. At least one EDS for the Communication Adapter. There may be multiple types of communication adapters that can be used for any given chassis. This may include communication adapters for other CIP networks; chassis and I/O modules would still stay the same.
2. At least one Chassis EDS. Some modular systems may support multiple chassis sizes or types.
3. One I/O module EDS for every type of I/O module that may populate this chassis. The EDS may also indicate a set of chassis that these modules may populate, so that these EDSs can be reused if the modules also plug into other chassis.

These three EDSs are linked to each other by certain mechanisms that are described below. Every module EDS contains entries for an “electronic key” that corresponds to the chassis into which the module can be inserted (logically or physically). Data links between the communication adapter EDSs and the I/O module EDSs are performed through a proxying mechanism: The adapter EDS contains entries using keywords like `ProxyAssemN`, `ProxyParamN` while the I/O module EDSs contain the associated entries with the `ProxiedAssemN`, `ProxiedParamN` keywords.
All EDSs of this set must be registered with the configuration tool, but typically only the communication adapter(s) will show up in the hardware resource window of some tools.

EDS DETAILS

The description that follows will first look at a modular design without parameters in the I/O modules and later expand this general principle to include parameters in the I/O modules.

Modular system without configuration parameters in the I/O modules

The first EDS to consider is a sample chassis EDS:

```
[File]
...

[Device]
VendCode            = 10000;
VendName            = "Sample Vendor";
ProdType            = 100;    $ Any Device Type in the
                              $ vendor specific range
ProdTypeStr         = "Common Interfaces  - No Device Object";
ProdCode            = 1;
MajRev              = 1;
MinRev              = 1;
ProdName            = "Test Chassis";
Catalog             = "2000-Chassis";

[Modular]
DefineSlotsInRack = 20;  $ This chassis has 20 slots
```

These entries represent a virtual electronic key. Each chassis has its own key; it is virtual because it does not exist in the chassis itself, but only in the EDS

Figure 7: EDS for a Sample Chassis

As the chassis itself is only a “container” without any I/O or parameters of its own, no further definitions are required. The chassis EDS shall not contain any I/O definitions.

The second EDS to consider is the communications adapter EDS:

The communications adapter module EDS serves several purposes:

- It defines how many chassis slots it occupies: `[Modular]` section, `Width` keyword
- It defines which slot in which chassis it may occupy: `[Modular]` section, `RackN` keyword. An adapter in a non-CIP chassis must always sit in slot 0. Non-CIP chassis are those chassis that do not use a protocol from the CIP family for communication between the communication adapter module and the I/O modules.
- It may define how a configuration tool can get ID information on the other (I/O) modules that sit in the same non-CIP chassis. This is done via the `[Modular]` section, `Query` keyword. This `Query` statement links to a data array that describes every module sitting in the chassis. The width of this data array is equal to the number of bytes that describe the external ID of a module; up to 16 bytes are allowed. The length of this data array is given through the number of slots in the chassis, not counting slot 0 (the adapter slot). This attribute (data array) is typically placed in a vendor specific object/attribute. The external ID is defined in each of the EDSs of the I/O modules that sit in this chassis. How this information is extracted from the I/O modules and written into this array is up to the developer of the system; DeviceNet only requires having the results addressable and readable. Adapters without query support are legal, but they miss many of the good features that modular EDSs can offer, e.g. they do not allow a configuration tool to read which modules are actually present in the chassis.
- It defines the I/O connection(s) to DeviceNet. As a consequence it must contain an `[I/O Info]` section. The entries in this I/O Info section point to assemblies so that the size of the I/O data can reflect the actual assembly of data coming from the individual I/O modules.
- It defines how the I/O data of the adapter (if any) and the I/O data of the other modules in the chassis are assembled into an input assembly and an output assembly: see `[Assembly]` section. There are several assembly types that can be used to accommodate the various assembly principles that may be used by a communication adapter (compressed, padded etc.) [3]. Collecting I/O data from the I/O modules is described by `ProxyAssemN` keyword entries that match the corresponding `ProxiedAssemN` keyword entries in the I/O module EDSs.
- It defines how parameter values are communicated to the I/O modules in the chassis. In the step-by-step approach of this paper, configurable parameters are omitted in this first example. Details on how to include parameters are described later in the section on parameters.

A typical adapter module EDS (without parameters in the I/O modules) looks like this:

```
[File]
...
[Device]
VendCode     = 10000;
VendName     = "Sample Vendor";
ProdType     = 12;
ProdTypeStr  = "Communications Adapter";
ProdCode     = 1;
MajRev       = 1;
MinRev       = 1;
ProdName     = "Sample Adapter Module";
Catalog      = "2000-Adapter";

[Modular]
Width        = 1; $ This module occupies 1 slot
Rack1        = 10000,100,1,1,1, , , ,0;
Query        = "20 01 24 01 30 64", 0x0E, 1, "00";

[IO_Info]
Default      = 0x01;
PollInfo     = 0x0001,1,1;
Input1       = ,0,0x0001,"Input Data",,Assem1,;
Output1      = ,0,0x0001,"Output Data",,Assem2,;

[Assembly]
Assem1       = "Total Input Data Size",,,,,,8,,,ProxyAssem1;
Assem2       = "Total Output Data Size",,,,,,8,,,ProxyAssem2;
ProxyAssem1  = "Exp. Mod. Inp. Data",,,0b0110,,,ModuleMemberList;
ProxyAssem2  = "Exp. Mod. Outp. Data",,,0b0110,,,ModuleMemberList;
```

Explanations see text

1 2 3 4 5

Figure 8: EDS for Sample Communication Adapter Module

Explanation of details in Figure 8:

1. `RackN` entry:
This is where the connection to the chassis EDS is made through the electronic key, for syntax see [1] chapter 7-3.6.1. The first 5 fields refer to the chassis described above in Figure 2. Fields 6, 7 and 8 are reserved, field 9 contains the only legal slot for this module (0). Multiple `RackN` entries may exist if this module can be used in more than one chassis.

2. `Query` entry:
The first field contains a path description to the data array that contains module IDs (external IDs). In this example, it's the Identity Object, instance 1, attribute 100. The second field contains the service code (0x0E) to get the data (get_attribute_single in this example). The third field defines the number of bytes per module (width of the array). The fourth field defines the external ID code that is used to describe an empty slot.

3. `InputN, OutputN` entries:
This entry follows the specification of the `InputN` keyword of the DeviceNet specification with the following extension as specified in [4]:

- The first field (size) is left blank (nothing else makes sense since the I/O size is not known in advance).
- The second field (number of significant bits) is set to 0 (nothing else makes sense since the I/O size is not known in advance).
- Instead of pointing to a connection path in fields 5 and 6, the name of an assembly from the `[Assembly]` section is given in field 6.

4. `AssemN` entries:
These two entries follow the syntax of an `AssemN` entry as defined in [1] chapter 7-3.5.7. Fields 2-6 are left empty. Fields 7 and above contain assembly member entries. In this case, field 7 contains the number of input/output bits of this adapter module, field 8 is empty. Field 9 is empty, therefore field 10 contains the reference to the `ProxyAssem1/2` entries described below.

5. `ProxyAssemN` entries:
These keywords define the sum of the I/O sizes of the modules in the chassis. They follow the syntax of the `AssemN` keyword with the modular enhancements defined in chapter 7-3.6.2.3 of [1]. Field 4 now contains the newly defined Assem type, see [3]. In this case it is set to type 3, variable size, compressed, adapter not included; this means that every module may vary in size, empty slots do not contribute any data to the assembly and no adapter data is included. More on Assem types in the appendix. The last field (ModuleMemberList) "collects" all the I/O data defined in the `ProxiedAssemN` keyword of each of the I/O modules in the chassis. In this case, `ProxiedAssem1` describes input data and `ProxiedAssem2` defines output data.

There are many variations of the type of assembly that can be used and a description of these types would go beyond the scope of this paper. A few more details are shown in the appendix.

Finally, I/O module EDSs:

Every I/O module type in the chassis also needs an EDS of its own that describes its characteristics. The following features are covered by this EDS:

- It defines how many chassis slots it uses: `[Modular]` section, `Width` keyword.
- It defines which slots in which chassis it may occupy: `[Modular]` section, `RackN` keyword. An I/O module in a non-CIP chassis must always sit in a slot other than 0. I/O modules that may be used for multiple chassis have one `RackN` entry for every chassis they may populate.
- It defines a unique identity (the External ID within the chassis it may populate) of the I/O module. In a non-CIP chassis, this ID can be retrieved through the data array defined in the Query entry of the adapter EDS.
- It defines module I/O data assemblies that are assembled into overall node assemblies by the adapter module. These assemblies (defined by the `ProxiedAssemN` keywords) have to match what is defined by the `ProxyAssemN` keywords in the adapter module.
- It defines module parameters and their use, more on that in the section on parameters in modular EDSs.

Here is a simple I/O module EDS:

```
[File]
...

[Device]
VendCode          = 10000;
VendName          = "Sample Vendor";
ProdType          = 101;      $ Any Device Type in the
                              $ vendor specific range
ProdTypeStr       = "Modular Discrete I/O";
ProdCode          = 15;
MajRev            = 1;
MinRev            = 1;
ProdName          = "Test Expansion I/O Module";
Catalog           = "2000-I/O";

[Modular]
Width             = 1;  $ This module occupies 1 slot
Rack1             = 10000,100,1,1,1,,,,1,2,3,4,5,6,7,8,9,
                    10,11,12,13,14,15,16,17,18,19;
ExternalID        = "01";

[Assembly]
ProxiedAssem1     = "Input data Array",,,,,,16,;
ProxiedAssem2     = "Output data Array",,,,,,8,;
```

Explanations see text

1 2 3

Figure 9: Expansion I/O Module EDS

Explanation of details in Figure 9:

1. `Rack1` entry:
Every I/O module needs at least one Rack entry. There may be multiple Rack entries if the module can be used in more than one chassis. This entry follows the same syntax as in the adapter EDS, but there are typically multiple legal slots, 19 in this example (all but slot 0).

2. `ExternalID` entry:
This is the identification of this module that can be accessed through the Query definition in the adapter EDS.

3. `ProxiedAssemN` entries:
In this example, there is a total of 2 input bytes and 1 output byte. These two entries follow the syntax of an `AssemN` entry as defined in [1], chapter 7-3.5.7. Fields 2-6 are left empty. Field 7 contains the number of input bits of this I/O module, entry 8 is empty. The early versions of configuration tools typically only use full bytes for I/O sizes, i.e. any number of bits is rounded up to the next multiple of 8.

The above is thus the complete set of EDSs that is required to describe a modular device without configurable parameters in the I/O modules. When the above I/O modules sit in a chassis together with a communication adapter as defined above, the total input data size will be 1 byte for the adapter plus 2 bytes for every I/O module and the total output data size will be 1 byte for the adapter plus 1 byte for every I/O module. Other types of modules in the same chassis may contribute other amounts of data.

Independent of DeviceNet, the developer has to provide the following functionality (summary of what is described above):

- An attribute (somewhere in a vendor specific object/attribute) in the communications adapter with an array of module IDs.
- A mechanism that reads all the module IDs from the I/O modules across the chassis backplane and writes them into this array.
- A mechanism that builds an input assembly and an output assembly from the I/O data provided by each of the I/O modules. This mechanism must follow the rules of the algorithm described in the EDS and executed in the configuration tool, i.e. communication adapter data (if any) first, followed by the I/O data of the modules in the chassis. Fractional byte I/O sizes must be rounded to the next full byte before they are added to the assembly.
- It is important to note the relationship between the input/output entities of the adapter and the actual modules:
- Input1 → Assem1 → ProxyAssem1 → ProxiedAssem1 and
  Output1 ← Assem2 ← ProxyAssem2 ← ProxiedAssem2

Modular system with configurable parameters in the I/O modules

Apart from configurable parameters in the adapter itself that follow the normal parameter functionality, it is possible to have configurable parameters “sitting” in the I/O modules that are “proxied” by the adapter. There are two different concepts that can be implemented:

1. The overall number and meaning of all configurable parameters is known. The adapter provides access to all possible configurable parameters. Individual I/O modules implement subsets of this overall parameter set.
2. The overall number and meaning of all the configurable parameters is not known. The adapter will only provide “templates” that will be used by the individual parameters of the I/O modules. The maximum number of configurable pa-

rameters of this type in any given I/O module must not exceed the number of "templates" provided in the adapter. Early versions of configuration tools may not support both methods.

Methods #1 and #2 may be combined in any given system once the functionality of method #2 is implemented in the configuration tool.

Let's first have a look at method #1:

The adapter EDS contains a set of proxy parameters that are listed in the EDS with the `ProxyParamN` keyword. The I/O modules use a subset of these parameters by listing them under the `ProxiedParamN` keyword in their individual EDSs, see examples below:

The adapter EDS defines the full set of proxy parameters (2 in this example):

```
[Params]
ProxyParam4       = 0,6,"20 64 24 SLOT 30 04",,
      0xC7,2,"Size","mm","",Module,Module,Module,,,,,,,,;


ProxyParam5       = 0,6,"20 64 24 SLOT 30 05",0x02,
      0xC7,2,"Color","","",Module,Module,Module,,,,,,,,;
```

Figure 10: ProxyParamN Entries in a Communication Adapter EDS

Explanation of details in Figure 10:

`ProxyParamN` entries:
The "`SLOT`" keyword in the path field is used to point to the individual slots that may be populated by I/O modules (1 through 19 in this example). The "`Module`" entries are placeholder for the min, max and default values that may be defined in the individual I/O modules. When the "`Module`" entry is present, it means that the value for this field is to be obtained from the corresponding field in the `ProxiedParamN` entry of the EDS for the I/O module. Without the "`Module`" placeholder, any field defined in the adapter EDS will be the same for all modules. With early versions of the configuration tool(s), "`Module`" placeholders might only be supported for the min, max and default fields.

The I/O module EDS uses all of the proxy parameters defined in the adapter EDS or a subset of only those parameters that are actually used in this type of I/O module. The example below shows the full set to illustrate the use of enumeration. If enumeration is used it may be written into the adapter module EDS (identical enumeration for all modules) or into the I/O module EDS:

```
[Params]
ProxiedParam4      = ,,,,,,,,,0,20,5,,,,,,,,;

ProxiedParam5      = ,,,,,,,,,0,4,1,,,,,,,,;

ProxiedEnum5       = 0,"red",1,"green",2,"yellow",
                     3,"pink",4,"blue";
```

Figure 11: ProxiedParamN Entries in an I/O Module EDS

Explanation of details in Figure 11:

In the `ProxiedParam4` entry, everything but the min, max and default values are already defined by the `ProxyParam4` in the adapter EDS.
The `ProxiedParam5` entry is similar to `ProxiedParam4`, but this shows how enumeration is used. It may be different for every type of module used.

Any parameter in the adapter (proxy parameter) that is not matched by a parameter in the I/O module (proxied parameter) and vice versa is not configurable and will not be displayed in the configuration tool.

Method #2 is not totally different, but it means that more fields in the `ProxyParamN` entry are referred to the individual I/O module's `ProxiedParamN` entry, e.g. Parameter Name or Units String. Since this method is not yet implemented in any known configuration tool, no further details are described in this paper.

To accommodate configuration parameters in the individual modules plugged into the chassis of a modular DeviceNet device, the following structure (see Figure 12) must be provided in the adapter on top of what is needed to transmit I/O data, based on the functionality of the current version of the configuration tool:

- An array of attributes, typically created in a vendor specific object:

  - This object has as many instances as there are slots in the chassis.
  - The number of attributes in this object is equal to the superset of parameters of all modules combined.
- Parameters that are identical (name and data size) in two or more modules can be handled through the same attribute.
- Parameters that are not identical need to be represented by individual attributes.
- The resulting number of attributes may thus be much larger than the maximum number of parameters in any given module.
- The adapter must have a mechanism to exchange data between this array of attributes and the individual modules based on the list of parameters supported by each module.

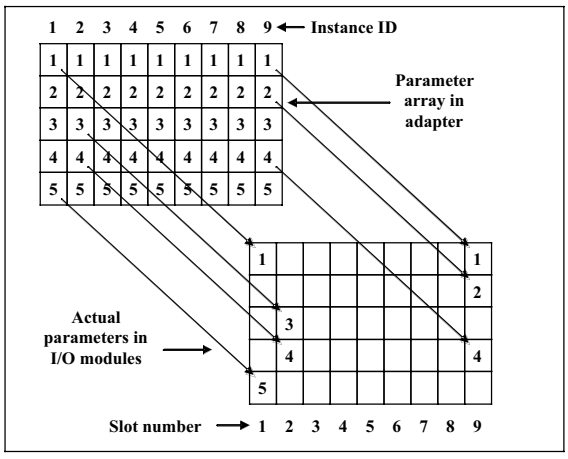

Figure 12: Relationship between Parameters in Adapter and I/O Modules

Under certain conditions, the required data array can be reduced in size by creating multiple `ProxyParamN` entries in the adapter EDS pointing to the same location, but only using one of these entries through a `ProxiedParamN` entry in the module EDS. This will work as long as this set of `ProxyParamN` entries uses the same data type. The end result will be very similar to method #2 (where more fields are referred to the module EDS), but the adapter EDS will still have to have one `ProxyParamN` entry for every parameter used in the device.

CONCLUSION

The above described functionality extensions provide a very powerful enhancement to the configuration methods for DeviceNet products. In particular, the capabilities of modular EDSs go far beyond the original concept of EDS-based configuration and allow a functionality not found in any other fieldbus system. The paper above gives a description of the use of assemblies and of the general principle of modular EDSs. This extension to modularity opens a huge range of additional possibilities that go well beyond the scope of this paper. For further reading, a more complete description of further details is shown in the appendix.

REFERENCES

[1] CIP Common Specification, Release 1.0
[2] DeviceNet Specification, Release 2.0, Errata 5
[3] CIPSE-001-003, Adapter Rack/Assem Data Formats
[4] DSE-001-085, AssemN Keyword in IO_Info Section
[5] CIPSE-001-004, Assembly Size Controlled by Parameter
[6] CIPSE-001-001, EDS Internal Only Parameter Descriptor Bit

Rockwell Automation Germany
Düsselberger Str. 15
42781 Haan
Germany
Phone: +49-2104-960-193
Fax: +49-2104-960-197
Email: vschiffer@ra.rockwell.com
Website:
http://www.automation.rockwell.com

APPENDIX: Further Sophistication

I/O Module renumbering:

There may be cases where the natural slot numbering (starting at slot 0) may not be appropriate. To avoid user confusion, the following method as defined in paragraph 7-3.6.1 of the CIP Common spec can be used to create a new naming scheme for the slots:

```
[Modular]
DefineSlotsInRack        = 20;       $ This chassis has 20 slots
SlotDisplayRule          = Param1;

[Params]
Param1       = 0,,,0x02,0xC6,1,"Slot Naming Convention",
               "","",0,19,0,,,,,,,,;
Enum1 =
0, "Adapter",
1, "Slot 0",
2, "Slot 1",

...
19, "Slot 18";
```

Figure A 1: Example of renaming slots in a modular chassis EDS

In principle, any text can be placed into the enumeration, but it is advisable to keep it short to make it easier to read in the slot display column.

Further I/O Assembly Details:

There are various types of assemblies and several examples are described in [3]. To describe all of them in full detail will go beyond the scope of this paper. However, the following details of the use of `AssemN` and `ProxyAssemN` are described here, because they will often be used:

Variable I/O size modules:

[5] describes a method that allows settable I/O sizes within assemblies:

First of all, there must be a data structure that allows getting and setting the I/O size values for all modules, e.g. through the following definitions:

```
[Params]
ProxyParam4 = 0,6,"20 04 24 SLOT 30 10",,0xC7,2,"Input Size",
              "Bytes","",Module,Module,Module,,,,,,,,;
ProxyParam5 = 0,6,"20 04 24 SLOT 30 11",,0xC7,2,"Output Size",
              "Bytes","",Module,Module,Module,,,,,,,,;
```

Figure A 2: ProxyParamN Structure for Variable I/O Size

This structure must be mirrored by an equivalent structure in the I/O modules, e.g.

```
[Params]
ProxiedParam4 = ,,,,,,"","","",0,4,  2,,,,,,,,;
$ Input data size between 0 and 4 bytes, default   2 bytes

ProxiedParam5 = ,,,,,,"","","",0,  2,1,,,,,,,,;
$ Output data size between 0 and  2 bytes, default 1 byte
```

Figure A 3: ProxiedParamN Structure for Variable I/O Size

The I/O sizes set by this method can now be used in suitable `ProxyAssemN` statements in the adapter EDS:

```
[Assembly]
ProxyAssem1  = "Expansion Module Input Data",,ProxyParam4,
               0b0110,,,ModuleMemberList;
ProxyAssem2  = "Expansion Module Output Data",,ProxyParam5,
               0b0110,,,ModuleMemberList;
```

Figure A 4: ProxyAssemN Structure for Variable I/O Size

The corresponding ProxiedAssemN statements in the I/O module EDS will then look like this:

```
[Assembly]
ProxiedAssem1      = "Input Data Array",,,,,,,;
ProxiedAssem2      = "Output Data Array",,,,,,,;
```

Figure A 5: ProxiedAssemN Structure for Variable I/O Size

Text description of I/O assemblies:

As explained in the main text of this paper, RSNetworx for DeviceNet 4.00.00 and higher allows the association of names to I/O assemblies or part of these assemblies.

The same method can also be used for modular assemblies by placing assembly information in the I/O module EDS as follows:

```
[Assembly]
      ProxiedAssem1      = "Input data Array",,,,,,8,Param1,8,Param2;
...
 [Params]
      Param1       = ,,,0x0, 0xC6,1,"Input Data","","",0,0xFF,0,,,,,,,,;
      Param2       = ,,,0x0, 0xC6,1,"Status","","",0,0xFF,0,,,,,,,,;
```

Figure A 6: EDS Entries for I/O Assembly Description in Modular Device

The result will then look like this:

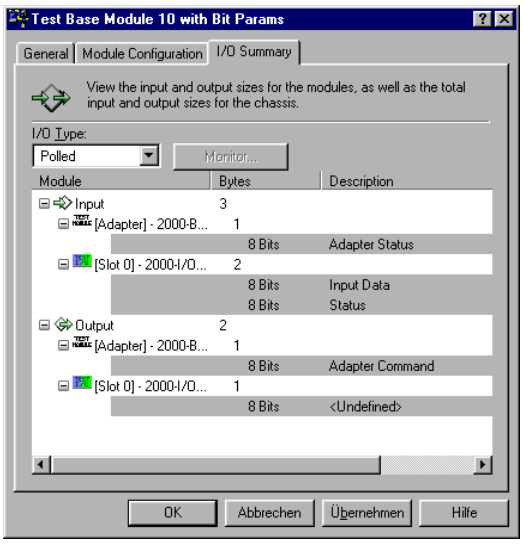

Figure A 7: I/O Assembly Display for Modular Device

Any assembly members that do not carry names defined by this method will be shown as <Undefined>.

Monitoring of I/O data:

RSNetworx for DeviceNet also allows displaying I/O data of assemblies by cyclically reading these values from the device. To do this, the assemblies described in `AssemN` or `ProxyAssemN` entries must contain a path to the data that is to be monitored. This is achieved by inserting another “layer” of assembly into the `[Assembly]` section of Figure 8, see Figure A 8.

```
[Assembly]
Assem1      = "Total Input Data Size",,,,,,8,,,Assem   3;
Assem2      = "Total Output Data Size",,,,,,8,,,Assem   4;

Assem3      = "Input Modules Data Size  ","20 XX 24 SLOT 30 XX ",
            ,0b0110,,,,ProxyAssem1;
Assem4      = "Output Modules Data Size  ","20 YY 24 SLOT 30  YY",
            ,0b0110,,,,ProxyAssem2;

ProxyAssem1 = "Exp. Mod. Inp. Data",,,,,,ModuleMemberList;
ProxyAssem2 = "Exp. Mod. Outp. Data",,,,,,ModuleMemberList;
```

Figure A 8: Path for I/O Data in Assembly

The `"20 XX 24 SLOT 30 XX"` and `"20 YY 24 SLOT 30 YY"` fields in the `Assem3` and `Assem4` entries point to the locations where the input and output data of the individual slots can be read.

Figure A 9 shows how monitored I/O data is displayed within the configuration tool. The module that was monitored was an Allen-Bradley 1798-IB4 (FlexArmor input module with 4 input points).

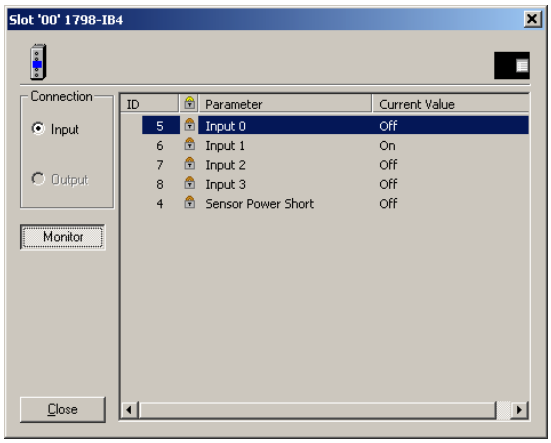

Figure A 9: Example of I/O Data Monitoring