

Autosar 4.0 - DataTypes

Technical Reference

Version 1.1

Authors	Thomas Bruni
Status	Released

Document Information

History

Author	Date	Version	Remarks
Thomas Bruni	11.11.2013	0.1	Document creation
Thomas Bruni	14.01.2014	0.2	Changes: 3.4 Platform types
Thomas Bruni	27.01.2014	0.3	Corrections in 3.4 Platform types
Thomas Bruni	29.01.2014	0.4	Creation of 2 new chapters: 3.4 Data type mapping 5.4 Data type mapping assistant 5.5 Type emitter
Thomas Bruni	21.02.2014	1.0	Release version
Thomas Bruni	14.03.2014	1.1	Changes for Mode Declaration Group mapping: chapters 3.4 and 4.4.

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	AUTOSAR_TPS_SoftwareComponentTemplate	4.2.0
[2]	AUTOSAR	AUTOSAR_SWS_PlatformTypes	2.5.0

Contents

1	Introduction	6
2	Data Types and Data Prototypes	7
3	Design of data prototypes in DaVinci tool chain.....	8
3.1	Data prototypes	8
3.2	Application data types	10
3.3	Implementation data types	11
3.4	Data type mapping	14
3.5	Platform types	15
3.5.1	Definitions	15
3.5.1.1	Autosar Standard Types	15
3.5.1.2	Platform Types	15
3.5.1.3	Valid C expression.....	15
3.5.2	Practice	16
3.5.2.1	Abstraction of SW from platform	16
3.5.2.2	Dependency between SW and platform	17
3.5.2.3	Platform types and Vector DaVinci Tool suite.....	18
3.6	Generation	22
4	Design examples	23
4.1	Primitive data element	23
4.2	Complex data element	36
4.3	Enumeration	48
4.3.1	Application level	48
4.3.2	Implementation level	52
4.4	Mode declaration.....	55
4.4.1	Mode declaration group in DaVinci Developer.....	55
4.4.2	Mode service port in BswM configuration	59
4.4.3	Mode request port and mapping.....	62
4.5	Implementation data type examples	66
4.5.1	Type reference	66
4.5.2	Value	67
5	Additional information	69
5.1	Compatibility and conversion.....	69
5.2	Measurement and calibration	70
5.3	Symbols	72
5.4	Data type mapping assistant	73

5.5	Type emitter	78
6	Glossary and Abbreviations	79
6.1	Glossary	79
6.2	Abbreviations	79
7	Contact.....	80

Illustrations

Figure 3-1	S/R port interface element	9
Figure 3-2	S/R port interface element init value	9
Figure 3-3	DaVinci Developer – workspace library.....	10
Figure 3-4	Application data type categories	10
Figure 3-5	Application data type Value property dialog	11
Figure 3-6	Implementation data type categories	12
Figure 3-7	Implementation data type Value property dialog	13
Figure 3-8	2 ways of modelling platform independent implementation data types	17
Figure 3-9	Modelling a platform specific implementation data type	18
Figure 3-10	Platform Types in DaVinci Developer	19
Figure 3-11	Implementation data type referencing a platform type.....	19
Figure 3-12	Native declaration: Autosar Standard Type	20
Figure 3-13	Native declaration: valid C expression	21
Figure 4-1	New Value application data type	24
Figure 4-2	My_TemperatureType	25
Figure 4-3	My_TemperatureType_CompuMethod	25
Figure 4-4	Celsius unit.....	26
Figure 4-5	Physical to Internal linear scale	26
Figure 4-6	Physical constraint.....	27
Figure 4-7	Type mapping set creation.....	28
Figure 4-8	Type mapping set naming.....	28
Figure 4-9	Add a data type mapping to a type mapping set	29
Figure 4-10	My_TemperatureInterface	30
Figure 4-11	My_TemperatureElement.....	30
Figure 4-12	SWC_Sender modeling	31
Figure 4-13	SWC_Receiver modeling.....	32
Figure 4-14	Connection of My_TemperatureInterface ports	33
Figure 4-15	Missing data type mapping error message	33
Figure 4-16	Reference a type mapping set in a SWC	34
Figure 4-17	My_SpeedType settings	36
Figure 4-18	New Record application data type	37
Figure 4-19	Record type creation with 2 record elements	38
Figure 4-20	New Record implementation data type	39
Figure 4-21	Set a record implementation data type.....	40
Figure 4-22	Record type mapping.....	41
Figure 4-23	Sender/receiver port interface with record element	42
Figure 4-24	Record manual init – part 1	43
Figure 4-25	Record manual init – part 2.....	44
Figure 4-26	Record constant creation.....	45
Figure 4-27	Referencing a record constant as init value	46
Figure 4-28	My_RecordInterface port connection	46

Figure 4-29	My_Enumeration type	48
Figure 4-30	My_EnumerationType_CompuMethod	49
Figure 4-31	Enumeration text table setting	49
Figure 4-32	Enumeration constraint	50
Figure 4-33	My_Enumeration mapping to uint8	50
Figure 4-34	"My_EnumerationInterface"	50
Figure 4-35	My_EnumerationImplType	52
Figure 4-36	My_EnumerationImplType_CompuMethod	52
Figure 4-37	My_EnumerationImplType_CompuMethod text table	53
Figure 4-38	My_EnumerationImplInterface	53
Figure 4-39	New mode declaration group	55
Figure 4-40	My_ModeDeclarationGroup	56
Figure 4-41	My_ModePortInterface	56
Figure 4-42	Mode ports connection	57
Figure 4-43	Mode port access	57
Figure 4-44	BswM configuration	59
Figure 4-45	BswM import in DaVinci Developer	59
Figure 4-46	Mode declaration group import in DaVinci Developer	60
Figure 4-47	ComM mode declaration group	60
Figure 4-48	My_ModeDeclarationGroup	62
Figure 4-49	My_ModeDeclarationGroup_IType	63
Figure 4-50	Mode declaration group mapping	63
Figure 4-51	My_ModeRequestInterface	64
Figure 4-52	My_ModeRequestPort	64
Figure 4-53	Type mapping set reference for mode request	64
Figure 4-54	New type reference	66
Figure 4-55	Value implementation data type	67
Figure 4-56	My_ImplValue_base base type	68
Figure 4-57	Implementation type constraint	68
Figure 5-1	Info Message #40286	69
Figure 5-2	Rte configuration for A2L generation	70
Figure 5-3	A2L generation folder	70
Figure 5-4	Symbol field for implementation data type	72
Figure 5-5	Data type mapping assistant opening	73
Figure 5-6	Data type mapping assistant	73
Figure 5-7	New mapping	75
Figure 5-8	Existing mapping	75
Figure 5-9	Created after new mapping	76
Figure 5-10	New mapping with implementation data type creation	76
Figure 5-11	Existing mapping with implementation data type creation	77
Figure 5-12	Created after new mapping with implementation data type creation	77

1 Introduction

This technical reference aims at presenting Data Type modeling with Vector DaVinci tool chain in Autosar 4.0.3 context.

2 Data Types and Data Prototypes

Embedded software uses data elements, which are entities containing information computed by algorithms, carried between application blocks, used as reference value, etc. Data elements are designed according to the information they need to represent. The structure definition of a data element is called data type.

In Autosar specification (see [1]) data elements are called data prototypes because they are the prototype or the instance of a certain data type that they reference. Data prototypes can be variable elements of a sender/receiver port interface, operation arguments of a client/server interface, modes of a mode switch interface, calibration parameters, per instance memory parameters, inter-runnable variables, etc.

Data prototypes may have different levels of representation. They may have a physical meaning, like speed information, or temperature information. They also must have an internal meaning which is used at code level, like 8 bit integer, or Boolean.

In order to carry these different levels of representation Autosar defines application data types and implementation data types. Application data types define data structure corresponding to the physical world. Implementation data types define data structure used in embedded code.

For a data prototype the application data type level is optional, whereas the implementation data type level is mandatory. Autosar allows 2 ways of modeling data prototypes:

- ▶ With a reference to an application data type:

In case a data prototype has a physical meaning, it shall reference an application data type. In that case, in order to also define the structure of the data prototype at code level, the application data type must be mapped to an implementation data type in a type mapping set.

- ▶ With a reference to an implementation data type:

If a data prototype does not have any physical meaning, it is possible to reference directly an implementation data type.



Note

During the design phase of a project, the architect models the data prototypes and maps them to data types. According to Autosar philosophy it is recommended to use as much application data types as possible and to reuse implementation data types in data type mapping. It allows staying at application level during design and avoids creating too many implementation data types in the code.

3 Design of data prototypes in DaVinci tool chain

Design is realized in DaVinci Developer. In the following chapters, figures and examples are presented from DaVinci Developer version 3.5.19 (SP1) and MICROSAR RTE 4.01.01.

3.1 Data prototypes

Data prototypes take several different forms in an Autosar ECU project. It can be an element of a sender/receiver interface, an argument of a client/server interface operation, an inter-runnable variable, a per-instance-memory variable, a calibration parameter... In general a data prototype contains the following attributes (see Figure 3-1 with S/R interface element example):

- Reference to an application data type or implementation data type:

This reference defines the type of the data element.

In case an application data type is chosen, this application data type must be mapped to an implementation data type in a type mapping set. This mapping set must be referenced by the application SWCs, which are using the data element.

In case an implementation data type is chosen, no mapping is necessary. The implementation data type will be used in the code as type of the data element.

- Measurement and calibration access:

In case the data element is intended to be accessed via measurement and calibration, the access type (ReadOnly or ReadWrite) shall be set here.

If no access is necessary, NotAccessible shall be set.

- Init value (not for per-instance-memory and not for operation arguments):

The init value field defines the initial value that the data element will take at startup of the software execution.



Note

The init value of a sender/receiver interface element is defined in the Com spec of each port prototype referencing the interface (see Figure 3-2).

- Invalid value handling (for sender/receiver interface elements):

It is possible to specify how invalid value shall be handled:

- None: no specific handling is defined
- Keep: an invalidate function is provided by the Rte to the SWCs that are sending the data element. The SWCs can report that the value is invalid by calling this function. The invalid value is kept, and a flag is set to inform the receivers that the value is invalid.
- Replace: an invalidate function is provided by the Rte to the SWCs that are sending the data element. The SWCs can report that the value is invalid by calling this function. The invalid value is replaced by the init value.

- Data constraints (for server/receiver interface elements):

- It is possible to define data constraints at data element level.

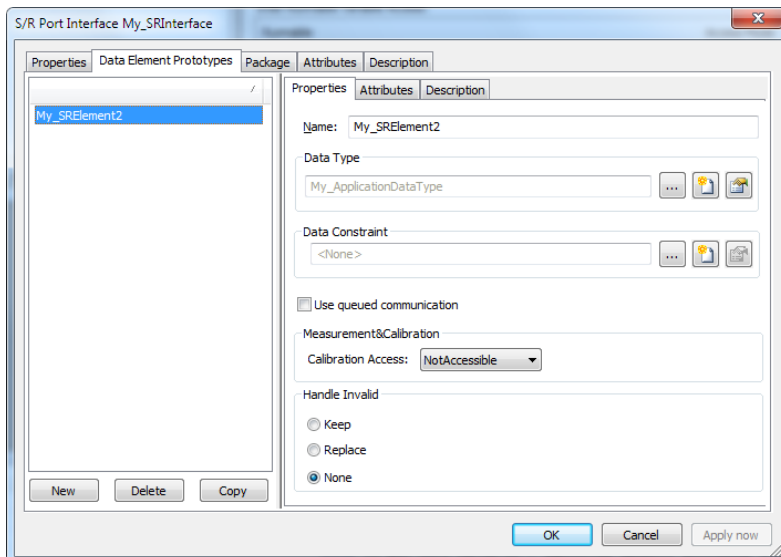


Figure 3-1 S/R port interface element

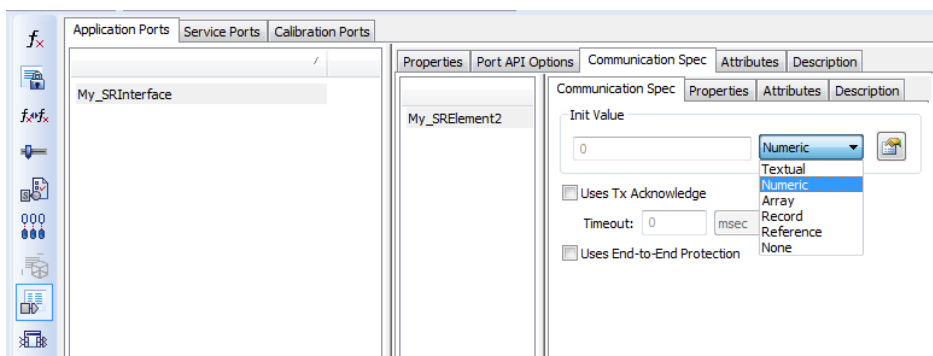


Figure 3-2 S/R port interface element init value

3.2 Application data types

In DaVinci Developer tool, it is possible to create application data types in the “Data Types” library under “Application Data Types” (see Figure 3-3).

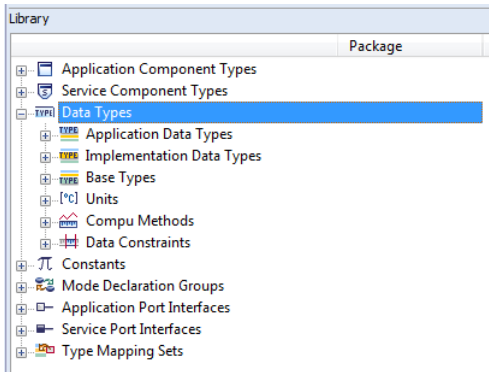


Figure 3-3 DaVinci Developer – workspace library

It is possible to choose the application data type category among Boolean, Value, String, Array and Record (see Figure 3-4).

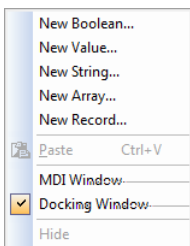


Figure 3-4 Application data type categories

An application data type contains several attributes (see Figure 3-5 a Value category example):

- Compu Method:

The computation method defines the conversion from internal representation to physical representation, in other words from code perspective to physical meaning.

It is possible to define linear, scale-linear, text table, scale-linear & text table conversions from internal to physical and/or from physical to internal.

- Unit:

The unit instance defines the physical unit.

**Note**

If a Compu Method is provided, it is recommended to set the unit in the Compu Method. If no Compu Method is provided, the unit can be entered directly in the application data type property dialog.

- Constraint:

With a constraint instance it is possible to define the internal and/or physical valid range of data values.

- Invalid value:

An invalid value may be defined. It however has no impact on the code. It is only set as information in DaVinci Developer. To invalidate a data element value, the sender SWC must call the invalidate function provided by the Rte.

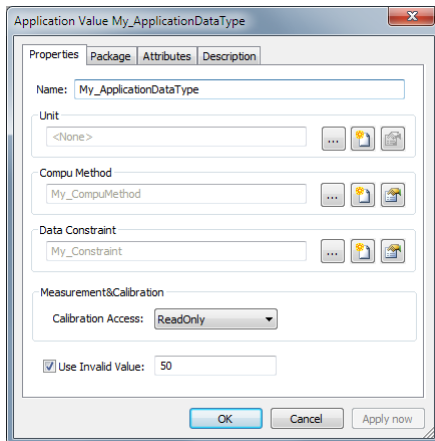


Figure 3-5 Application data type Value property dialog

More details and examples are presented in chapter 4.

3.3 Implementation data types

In DaVinci Developer tool it is possible to create implementation data types in the “Data Types” library under “Implementation Data Types” (see Figure 3-3).

It is possible to choose the implementation data type category among Value, Type Reference, Data Reference, Array and Record (see Figure 3-6).

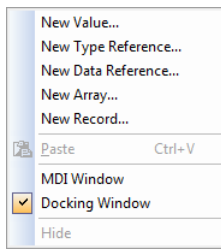


Figure 3-6 Implementation data type categories

An implementation data type contains several attributes (see Figure 3-7 with a Value category example):

- Base type reference:
 - This reference must be provided in implementation data types of category Value in order to define the corresponding base type.
 - Base types may be created manually in the “Data Types” library under “Base Types”.
 - It is also possible to link the implementation data type to an existing implementation data type (e.g. to a platform type) by choosing the category Type Reference.

**Note**

In case a new implementation data type is based on a type that does not exist, the designer may need to create a new base type.

If an existing type corresponds to the need of the designer, it is recommended to create a Type Reference and reference an existing type. In particular it is recommended to reference the platform types (see chapter 3.5).

- Compu method:

It is possible to define a computation method in the implementation data type.

**Note**

As defined by Autosar the compu method of an implementation data type is not intended to provide any physical meaning (this is the role of application data type). For an implementation data type, only TextTable conversion is allowed which generates enumeration strings in the code that are used instead of the numerical value.

- Constraint:

It is possible to define constraints at implementation level.

**Note**

A constraint at implementation level will be used generally to define the valid internal range.

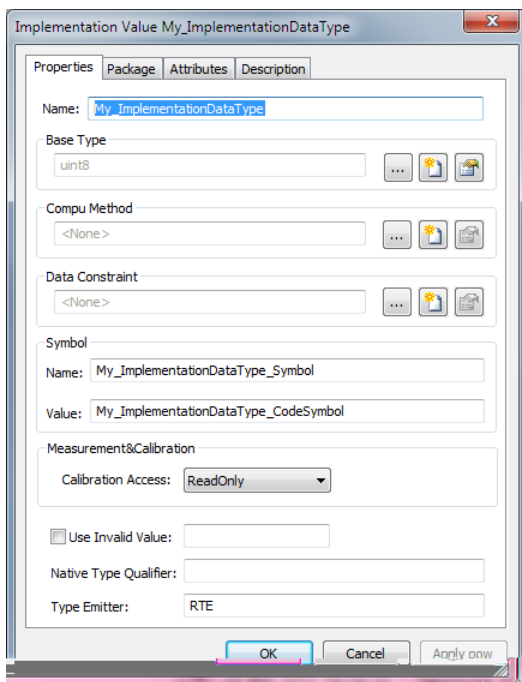


Figure 3-7 Implementation data type Value property dialog

3.4 Data type mapping

As previously written, application data types must be mapped to implementation data types. For this purpose data type mapping sets containing data type mappings must be created. Data type mappings contain two references, one to the application data type and one to the implementation data type.

In order to let the Rte generator access the right data type mapping for each data prototype needing a data type mapping, data type mapping sets must be referenced by the SWCs using the corresponding data prototypes.

Mode declaration groups may also be mapped to implementation data types in ModeRequestTypeMap contained also in data type mapping sets. The purpose of this mapping is not the representation of the mode declaration group in the code because the type used in the code for mode declaration groups is generated automatically by the Rte generator according to the number of mode declarations: uint8 for 1 to 256 mode declarations, uint16 for 257 till 65536 mode declarations, and so on...

The purpose of a ModeRequestTypeMap is to be able to create mode request ports. As specified by Autosar a component that requests mode changes to a mode master shall implement a sender/receiver P-Port on which the requests will be sent. The corresponding port interface's data element shall be typed by an implementation data type referencing a compu method containing an enumeration of the corresponding modes. That implementation data type shall be mapped to the corresponding mode declaration group.

Examples are presented in chapter 4.

3.5 Platform types

3.5.1 Definitions

3.5.1.1 Autosar Standard Types

Autosar Standard Types are the following symbols used in C code:

- > uint8
- > uint16
- > uint32
- > sint8
- > sint16
- > sint32
- > boolean
- > float
- > float32

3.5.1.2 Platform Types

Platform types are implementation data types which have an Autosar Standard Type as name.

Platform types, as subset of implementation data types, are part of the model.

3.5.1.3 Valid C expression

A valid C expression is a C type optionally combined to a C size qualifier and/or a C sign qualifier.

- > C types are: char, int, float, double, void
- > C size qualifiers are: short, long
- > C sign qualifiers are: signed, unsigned

3.5.2 Practice

Implementation data types need to be generated in C code and represented at CPU level. For this purpose 2 ways can be followed:

- ▶ With abstraction of SW from platform
- ▶ With dependency between SW and platform

3.5.2.1 Abstraction of SW from platform

This way of representing implementation data types at CPU level is advised, because it follows the Autosar philosophy of independency between SW and platform.

It consists in linking implementation data types to Platform Types, which are represented by Autosar Standard Types in C code.

Each implementation data type shall finally be linked to Platform Types. This may be done in 2 ways:

- ▶ The implementation data type's name is an Autosar Standard Type. In this case the implementation data type is implicitly the corresponding Platform Type.
- ▶ The implementation data type references a Platform Type. The implementation data is



Note

Platform_Types.h also contains 3 parameters which are CPU specific: CPU_BIT_ORDER, to define bit alignment, CPU_BYTE_ORDER, to define Byte order, and CPU_TYPE, to define the processor bit length.

Figure 3-8 summarizes these 2 ways of modelling:

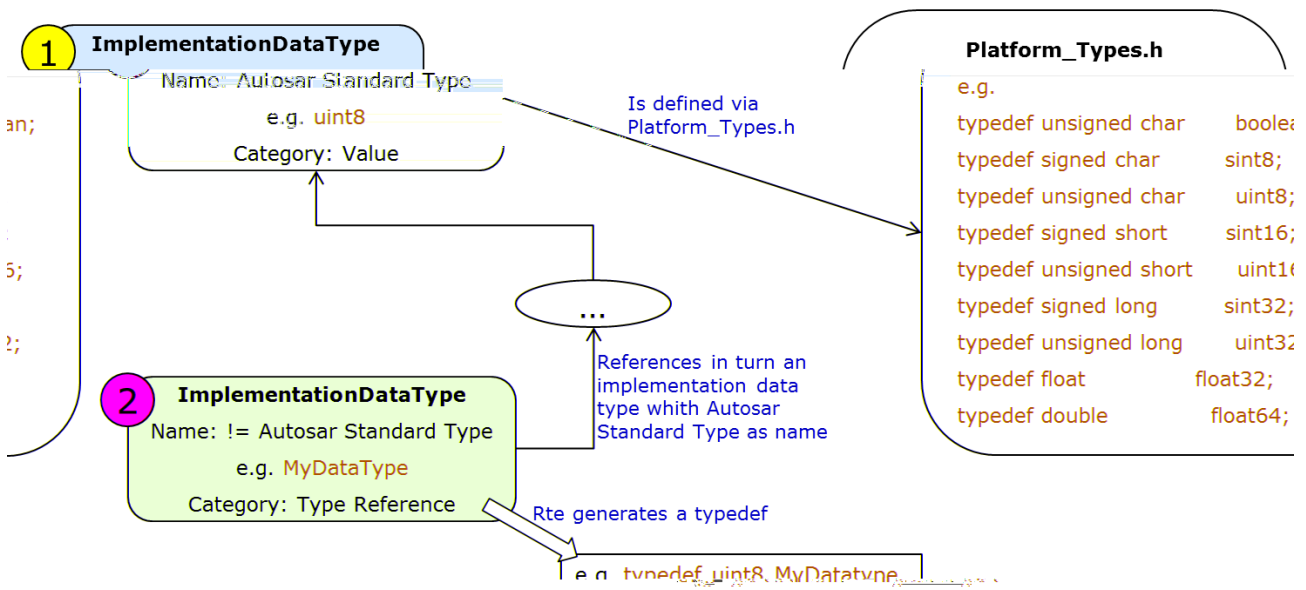


Figure 3-8 2 ways of modelling platform independent implementation data types

3.5.2.2 Dependency between SW and platform

In a same way as for Complex Device Drivers, Autosar gives also the chance to break this abstraction by setting CPU and/or compiler specific information in base types:

An implementation data type of category Value must reference a base type, which contains a native declaration parameter. Setting a valid C expression as native declaration will generate a typedef defining the implementation data type as this valid C expression in `Rte_Type.h`.

In this way, the implementation data type and the SWCs using this implementation data type are platform specific.

**Example**

Implementation data type name: MyImplementationDataType

Native declaration in base type: unsigned char

Generation: `typedef unsigned char MyImplementationDataType;`

Figure 3-9 summarizes this modelling:

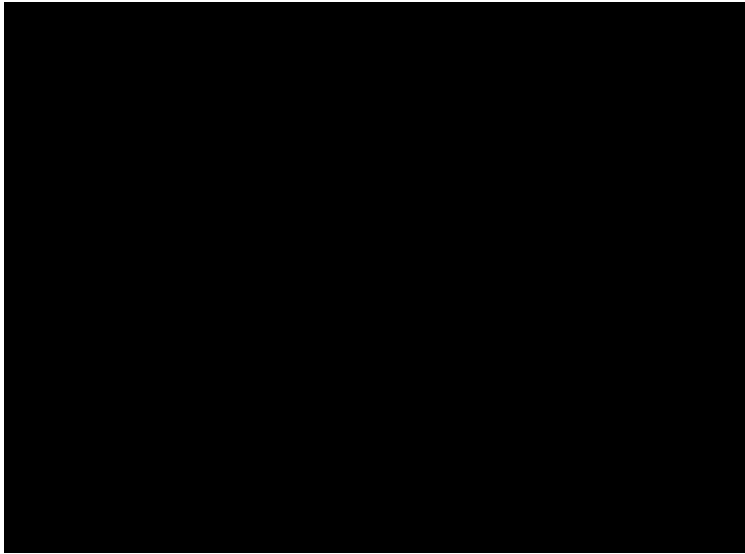


Figure 3-9 Modelling a platform specific implementation data type

3.5.2.3 Platform types and Vector DaVinci Tool suite

Vector DaVinci Developer provides the Platform Types in the Implementation Data Types library at workspace creation (see Figure 3-10).

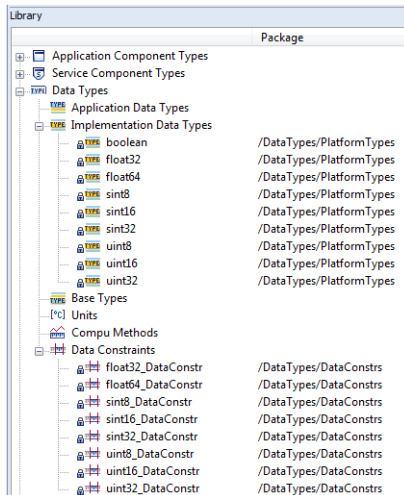


Figure 3-10 Platform Types in DaVinci Developer

According to Autosar it is advised to do the following:

- ▶ Use directly Platform Types as implementation data types.
- ▶ Create an own implementation data type of category “Type Reference” referencing a Platform Type:

Creating MyImplementationDataType as Type Reference pointing on uint8 (see Figure 3-11) Platform type will generate in Rte_Type.h:

```
typedef uint8 MyImplementationDataType;
```

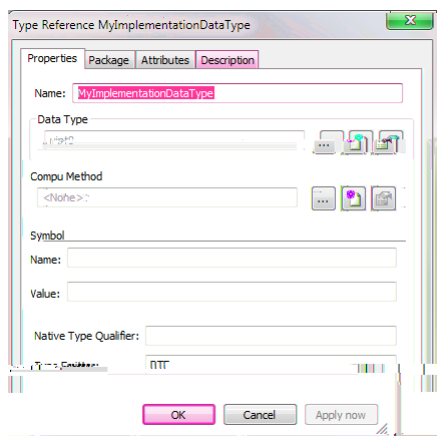


Figure 3-11 Implementation data type referencing a platform type

However other ways are possible:

- It is possible to write the name of an Autosar Standard Type as native declaration in the BaseType referenced by MyImplementationDataType of category Value (see Figure 3-12). This way is however not very pretty because its intention is to model a “Type Reference” without expressing it formally in the model. In this case the same typedef will be generated:

```
typedef uint8 MyImplementationDataType;
```

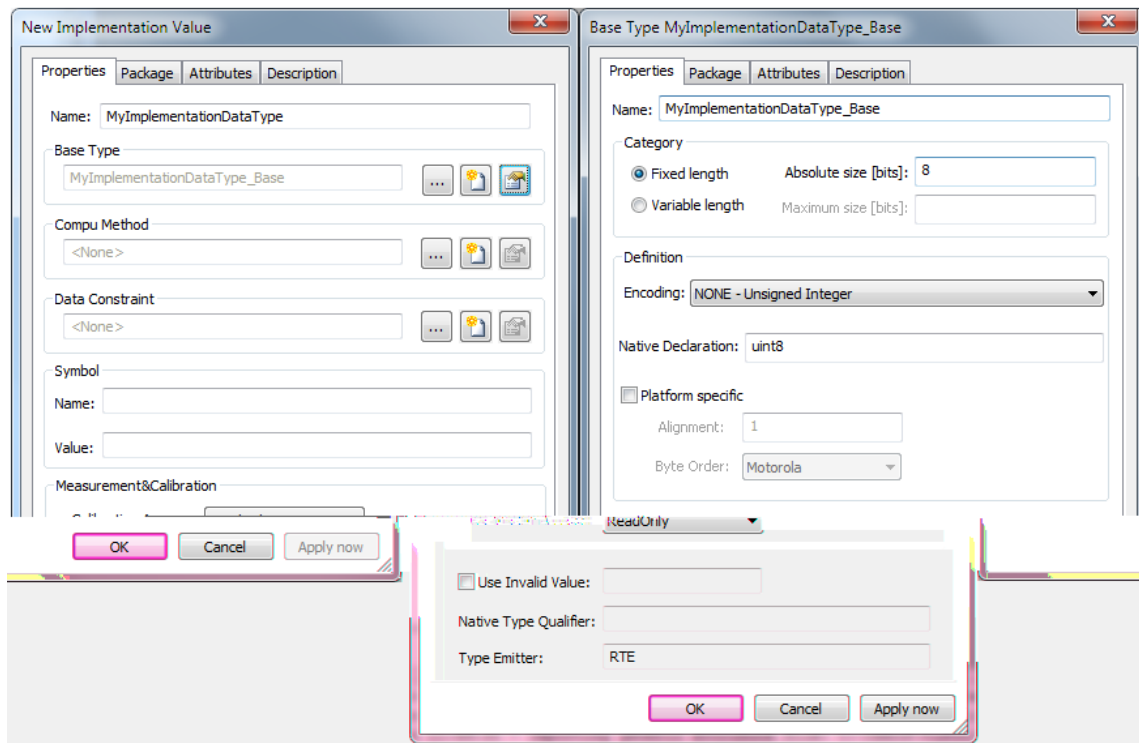


Figure 3-12 Native declaration: Autosar Standard Type

- BaseType with C native declaration: it is possible to generate the platform specific definition of an implementation data type using the native declaration field. In the provided example (see Figure 3-13), the following will be generated:

```
typedef unsigned char MyImplementationDataType;
```

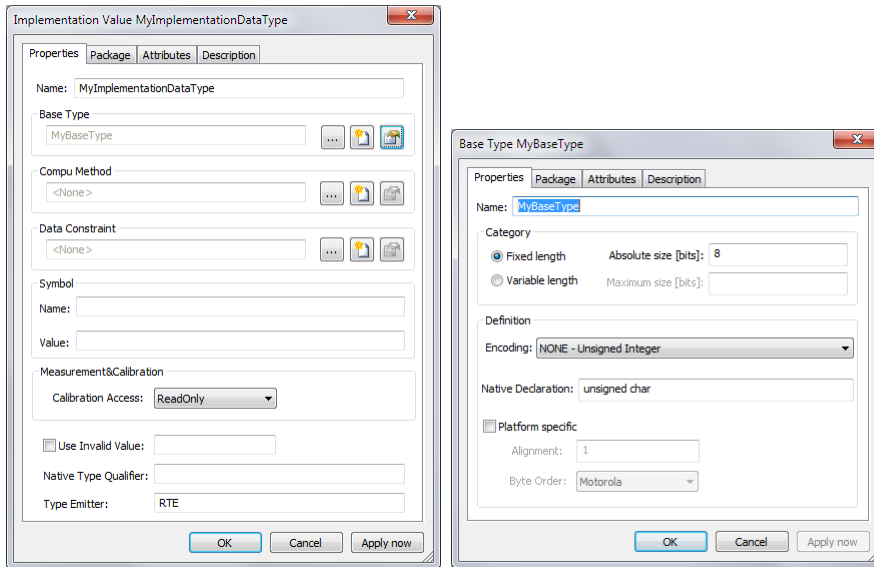


Figure 3-13 Native declaration: valid C expression

3.6 Generation

The following is generated out of the data type design:

- In code:
 - Implementation data types are provided in the generated code as typedef.
 - From application data types are generated:
 - Init values: they are calculated from the init value and the computation method settings.
 - Lower and upper range limits as defines: they are calculated out of the constraint range and the computation method.
 - Invalid value handling: the way of treating invalid values depends on the handling setting. The corresponding invalidate function is implemented in the Rte and provided to SWC that send the corresponding data element.
 - Enumeration: if the defined computation method is a text table, an enumeration is provided.
- For measurement and calibration:
 - An A2L file can be generated with data prototype access:
 - The read and write access is defined by the attribute specified at data element level. It overrides the attribute provided in application or implementation data types.

4 Design examples



Caution

Each ECU project is specific and has its own needs.

The examples presented in this chapter aim at showing how to use DaVinci Developer tool chain and what is generated out of the example configuration.

Not all possible configurations are presented here, and it is up to the ECU designer to choose the appropriate configuration settings depending on the needs of the project.

4.1 Primitive data element

This example shows the design of a primitive application data type referenced by a sender/receiver port interface data element.

A sender/receiver port interface is designed to send a temperature value in Celsius.

The physical values can be in the range $[-50^{\circ}\text{C}; +150^{\circ}\text{C}]$ with $0,5^{\circ}\text{C}$ resolution. In other words, the values can be -50°C , $-49,5^{\circ}$, ..., $+149,5^{\circ}\text{C}$, $+150^{\circ}\text{C}$.

In order to model the values at code level, a type covering such a range shall be defined. In this case it is necessary to cover 401 values. An 8 bit integer will not be sufficient because it covers only 256 values. A 16 bit integer (uint16 or sint16) can be used. The conversion function will be adjusted to this choice. In this example sint16 is chosen. A linear conversion from physical to internal with factor 2 and offset 0 ($f(x)=2 \cdot x+0$) will be used. The corresponding internal range will be $[-100; +300]$.

- ▶ An application data type is created to model the physical part of the data type:
 - > Creation of a new Value application data type:
 - > Right click on the “Application Data Types” library is done and “Value” is selected (see Figure 4-1).

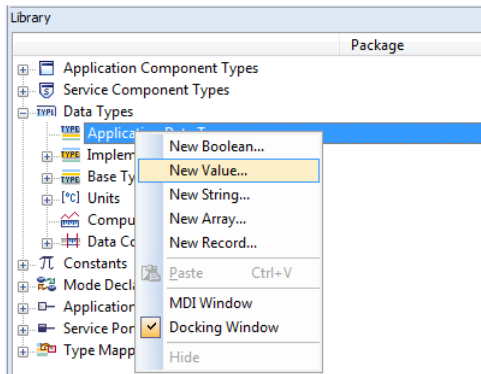


Figure 4-1 New Value application data type

- > The application data type attributes are filled (see Figure 4-1):
 - > A name is given: "My_TemperatureType"
 - > A computation method is created: "My_TemperatureType_CompuMethod" (see Figure 4-3):
 - Category is set to linear
 - A "Celsius" unit is created (see Figure 4-4) and is referenced in the compu method
 - "Physical to internal" conversion is chosen
 - The physical to internal linear scale is configured with factor 2 and offset 0 over the physical range [-50;+150] (see Figure 4-5)
- > A physical constraint is created: "My_TemperatureType_Constraint" (see Figure 4-6):
 - It defines the physical range with boundary inclusions: [-50;+150]



Note

The physical to internal conversion is also used for internal to physical conversion in case the defined function is invertible. It is the case in this example, so it is enough to specify only the physical to internal conversion.

The linear conversion factor and offset are used in the following way: With X_i representing the internal value and X_p the physical one.

- Physical to internal: factor F_p and offset O_p :

$$X_i = F_p \cdot (X_p + O_p)$$
- Internal to physical: factor F_i and offset O_i :

$$X_p = F_i \cdot X_i + O_i$$

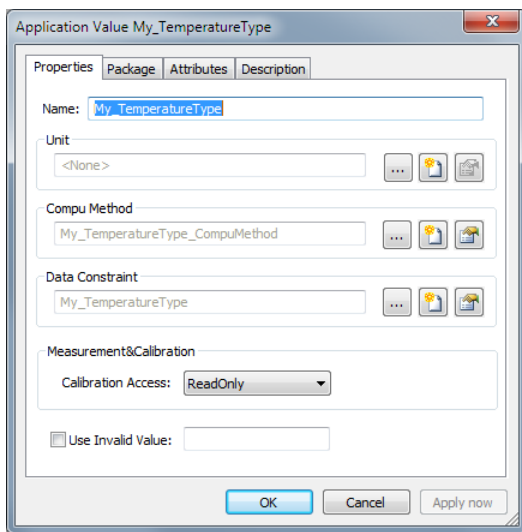


Figure 4-2 My_TemperatureType

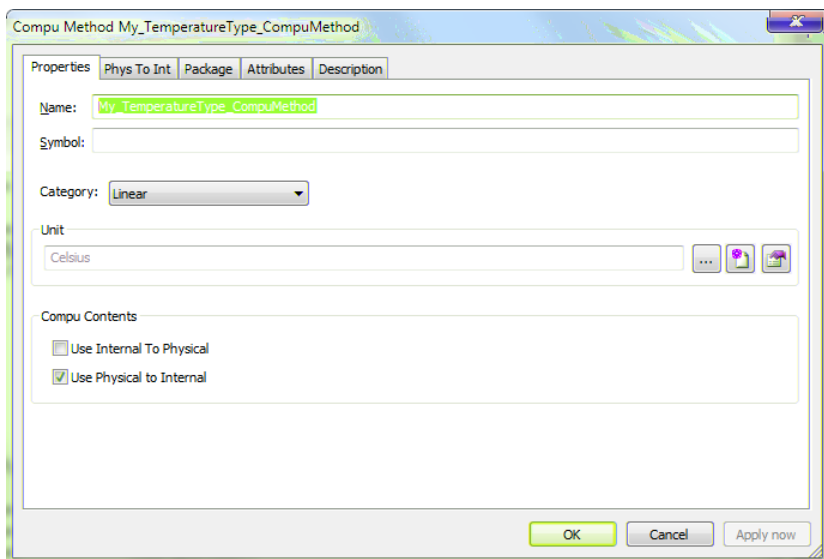


Figure 4-3 My_TemperatureType_CompuMethod

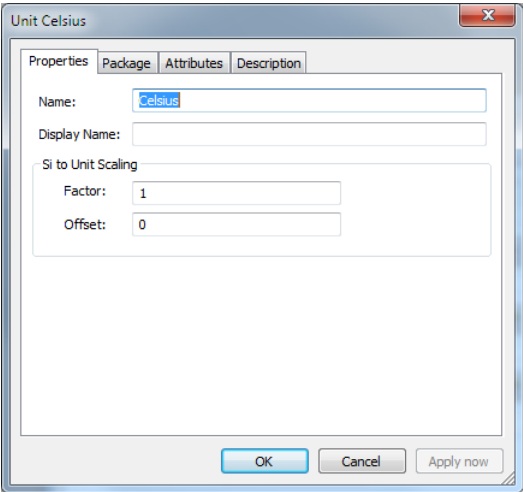


Figure 4-4 Celsius unit

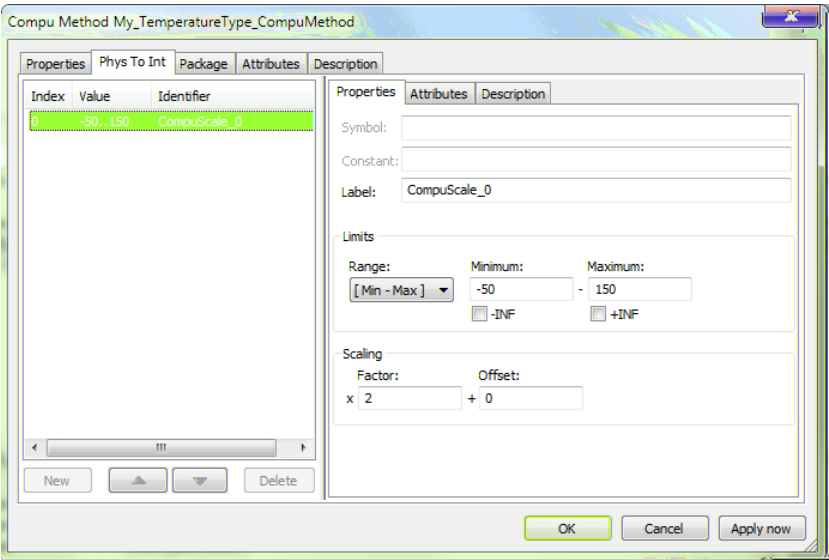


Figure 4-5 Physical to Internal linear scale

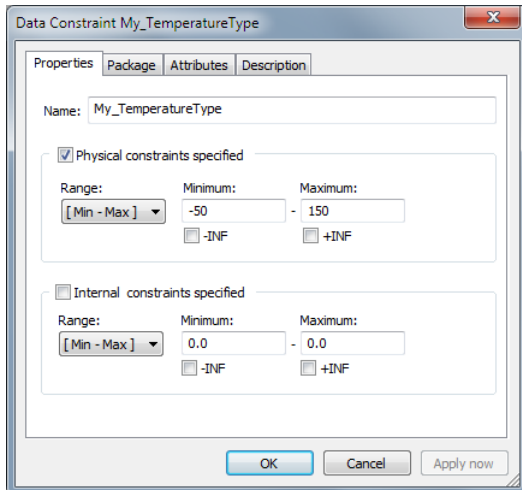


Figure 4-6 Physical constraint

- The internal part of the data type is modeled with the existing platform type sint16.

It is thus not necessary to create a new implementation data type.

- In order to finish the data type modeling, a data type mapping set has to be used to specify the mapping between “My_TemperatureType” and sint16.
- > A new type mapping set is created:
 - > Right click on the Type Mapping Sets library is done and “New Type Mapping Set...” is selected (see Figure 4-7).



Note

It is not necessary to create a new type mapping set for each data type mapping. A single type mapping set can contain several data type mappings.

It is even recommended to gather data type mappings in type mapping sets, not to have too many data type mapping set.

- > A name is given to the type mapping set (see Figure 4-8).
- > The data type mapping is done in the “Data Type Maps” tab of the type mapping set property dialog (the steps are described Figure 4-9)

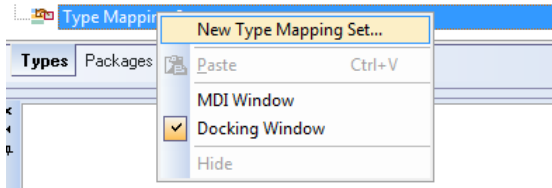


Figure 4-7 Type mapping set creation

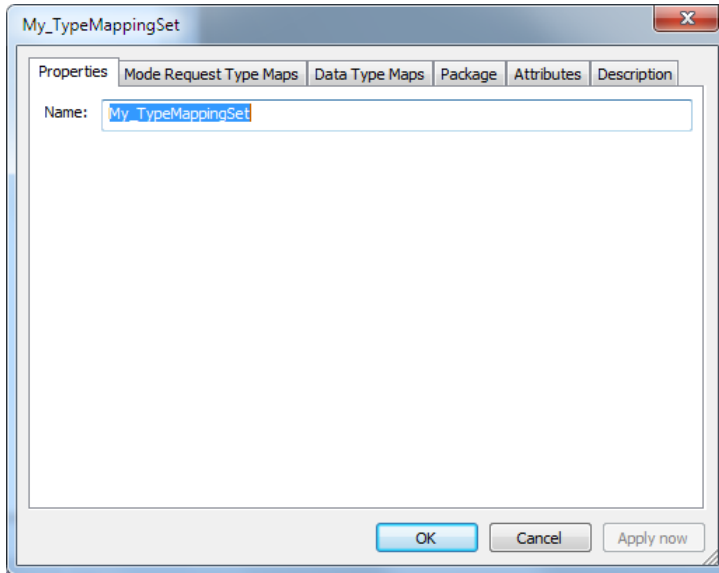


Figure 4-8 Type mapping set naming

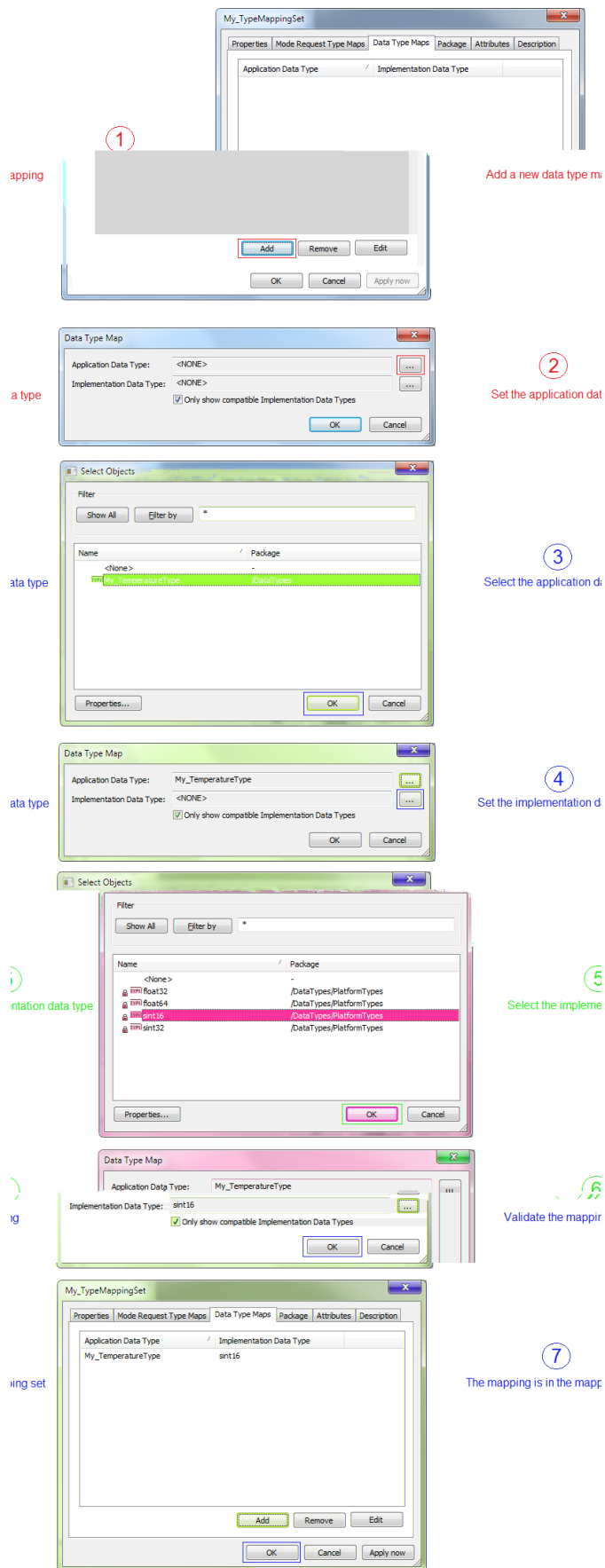


Figure 4-9 Add a data type mapping to a type mapping set



Note

In the Data Type Map dialog a check box is available to filter only the compatible data types. It is checked by default.

- Now that the data type for the temperature data elements is modeled, it can be used.

In this example a sender/receiver port interface containing a temperature data element is designed. The port interface “My_TemperatureInterface” is created (see Figure 4-10) and “My_TemperatureElement” is created inside (see Figure 4-11).

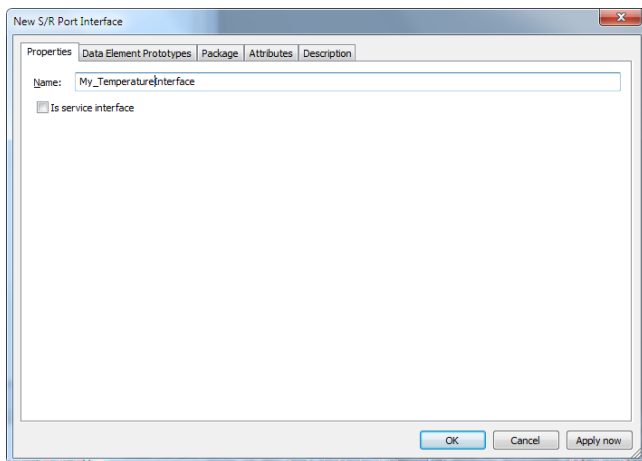


Figure 4-10 My_TemperatureInterface

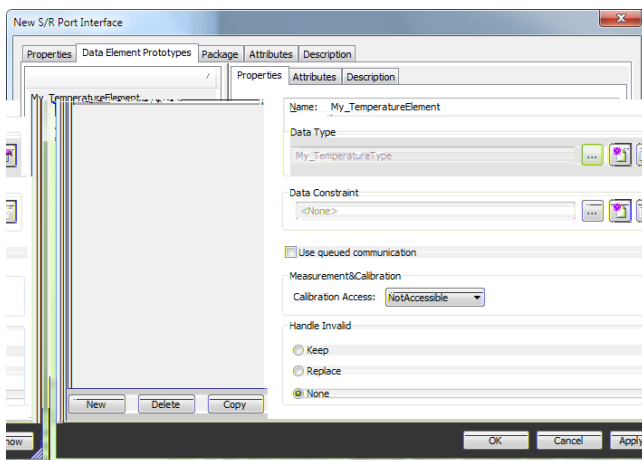
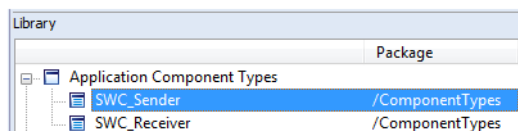


Figure 4-11 My_TemperatureElement

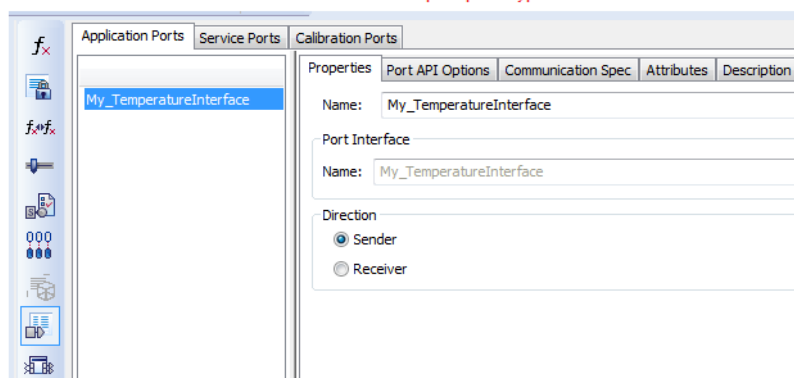
Two SWCs implementing port prototypes referencing this port interface are created. SWC_Sender is writing “My_TemperatureElement” in “SWC_SenderRunnable” and “SWC_Receiver” is reading it in “SWC_ReceiverRunnable” (see Figure 4-12 and Figure 4-13). The init value on both sides is set to 20.

SWC_Sender

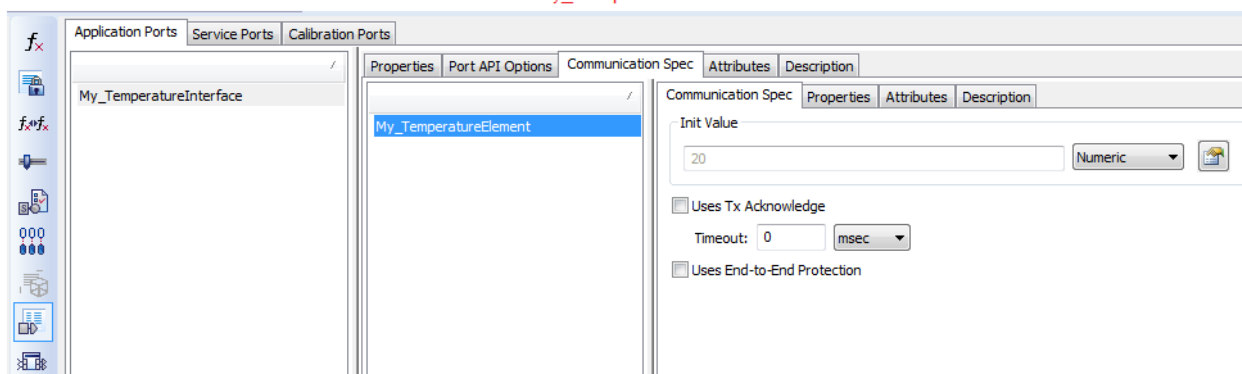
1: Create the SWC



2: Create the port prototype



3: Set My_TemperatureElement init value



4: Create the runnable and set the port access

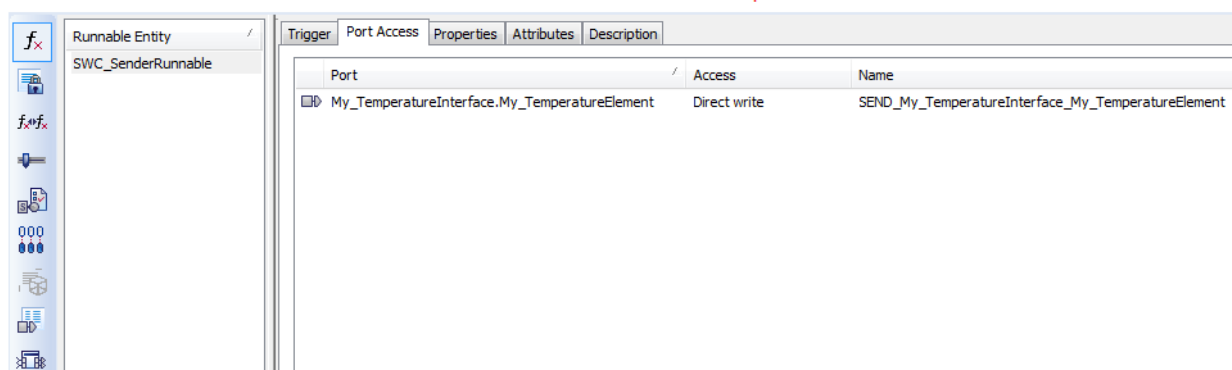
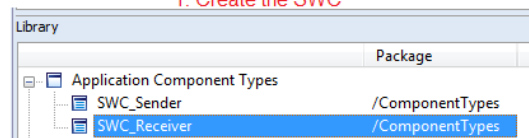


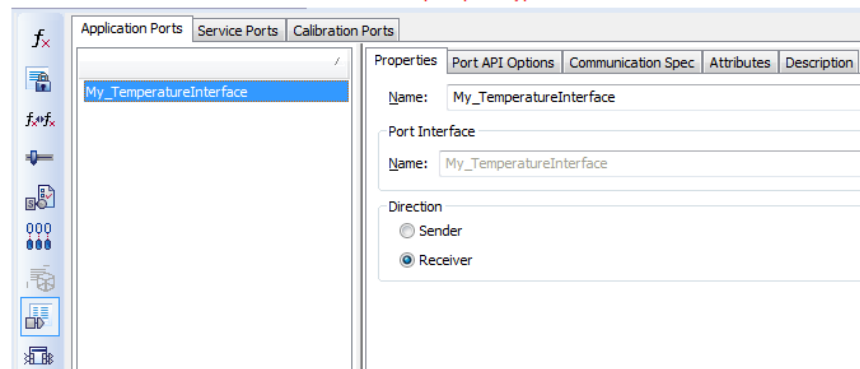
Figure 4-12 SWC_Sender modeling

SWC Receiver

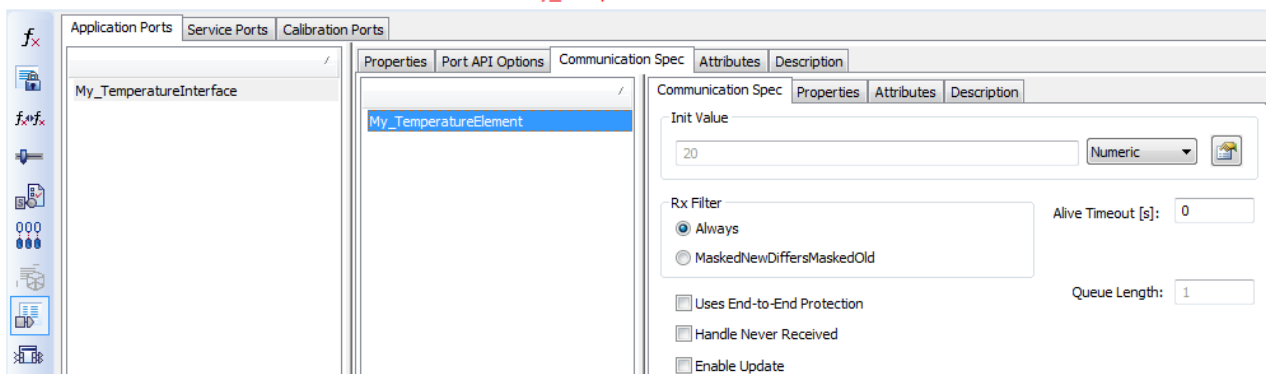
1: Create the SWC



2: Create the port prototype



3: Set My_TemperatureElement init value



4: Create the runnable and set the port access

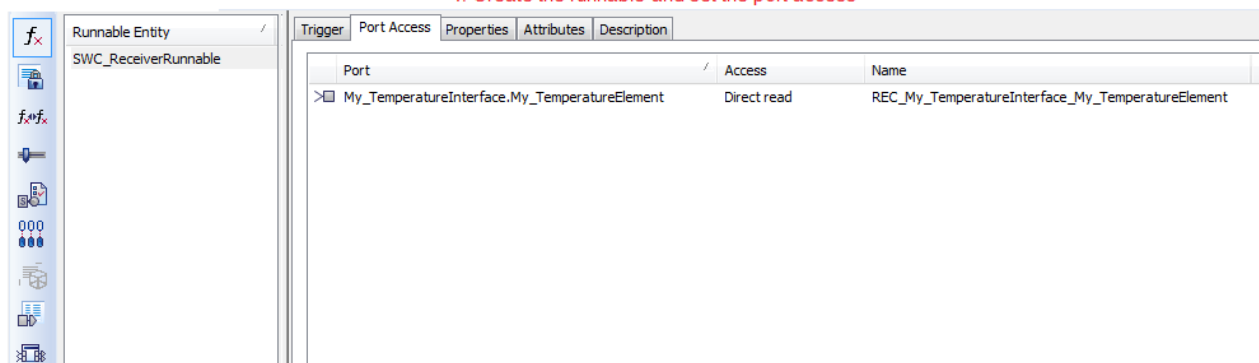
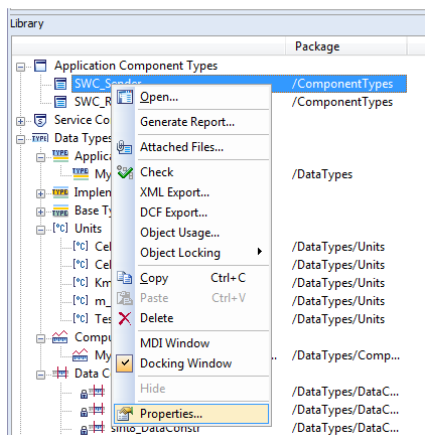
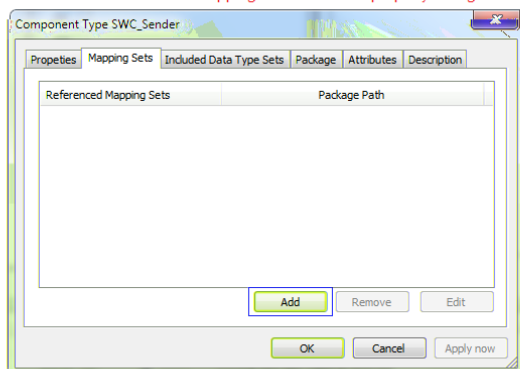


Figure 4-13 SWC_Receiver modeling

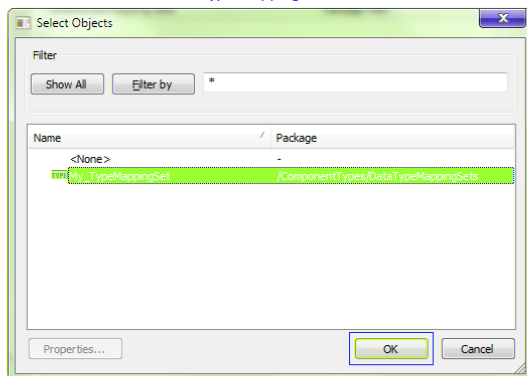
1. Right click on the SWC in the "Application Component Types" library and select "Properties..."



2. Select "Add" in the "Mapping Sets" tab of the property dialog



3. Select the type mapping set and click "Ok"



4. The type mapping set is now referenced by the SWC

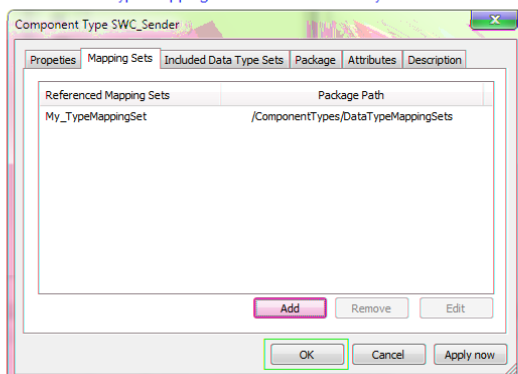


Figure 4-16 Reference a type mapping set in a SWC

- The system description resulting from the configuration done in DaVinci Developer is imported in DaVinci Configurator Pro. There the Rte generation and the SWC generation are performed.

The following code is generated:

In SWC_Sender.c:

```
Std_ReturnType Rte_Write_ ( data)
```

In SWC_Receiver.c:

```
Std_ReturnType Rte_Read_ ( *data)
```

In Rte_SWC_Sender.h and Rte_SWC_Receiver.h:

```
# define Rte_InitValue_ ( )
```

In Rte_SWC_Sender_Type.h and Rte_SWC_Receiver_Type.h:

```
# define _LowerLimit (-100)
# define _UpperLimit (300)
```

The data element of the read and write function is typed by sint16.

The init value is 40 (internal value of configured physical value 20).

The generated lower limit is -100 and upper limit 300.



Note

In the Rte generator version (4.01.01) the generated negative included limits are shifted by +1. If both limits are negative, the lower and upper limits are inverted.

These two issues are known and will be fixed in a future version.

4.2 Complex data element

This example shows the design of a complex application data type referenced by a sender/receiver port interface data element.

A sender/receiver port interface is designed to send a record containing a temperature value in Celsius as first element and a speed value in km/h as second element.

The temperature value is modeled as in chapter 4.1.

The speed physical values can be in the range [0 km/h; 300 km/h] with 0,5 km/h resolution. In other words, the values can be 0 km/h, 0,5 km/h,..., 299,5 km/h, 300 km/h.

In the same way as the temperature, speed needs an implementation type covering the possible values at code level. Uint16 is chosen. A linear conversion from physical to internal with factor 2 and offset 0 ($f(x)=2 \cdot x+0$) is used. The corresponding internal range will be [0; 600].

- A Value application data type ("My_SpeedType") is created and mapped to uint16 (see Figure 4-17). For details about primitive data type creation refer to chapter 4.1.

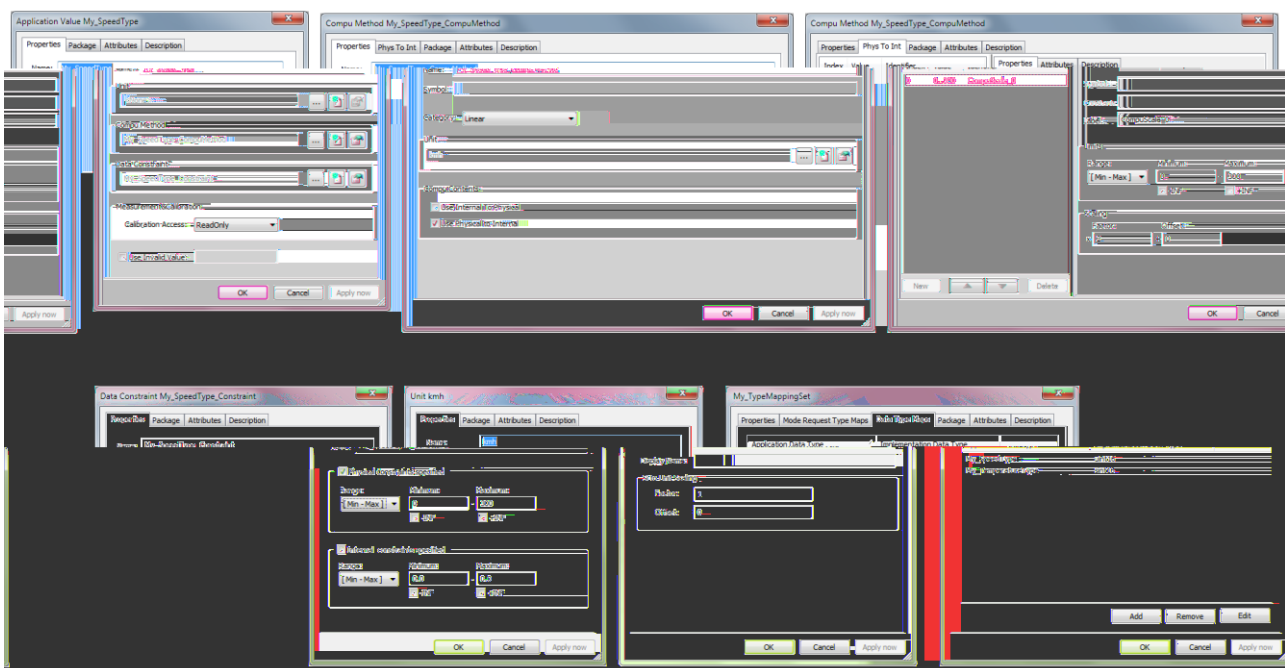


Figure 4-17 My_SpeedType settings

- A Record application data type ("My_RecordType") is created with a temperature element as first element and a speed element as second element:

- Right click on “Application Data Type” library is done and “New Record...” is selected (see Figure 4-18).

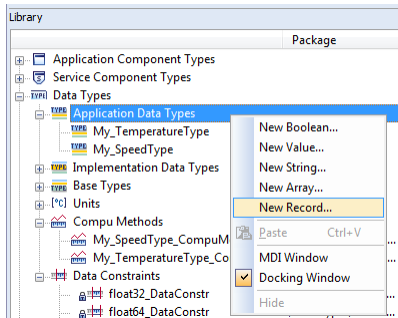
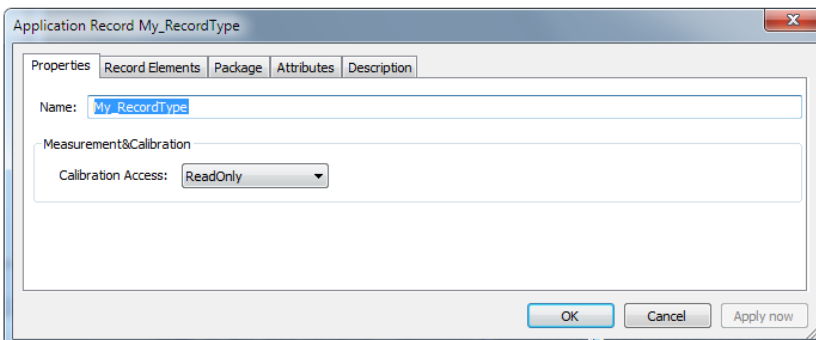


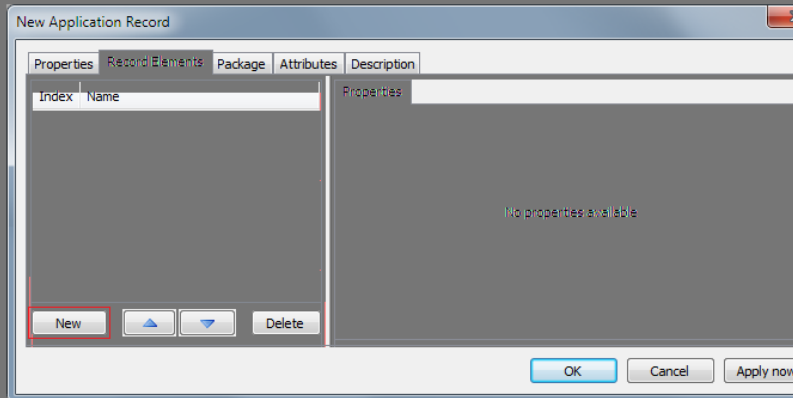
Figure 4-18 New Record application data type

- Enter the name of the record application data type and create 2 record elements (see Figure 4-19).

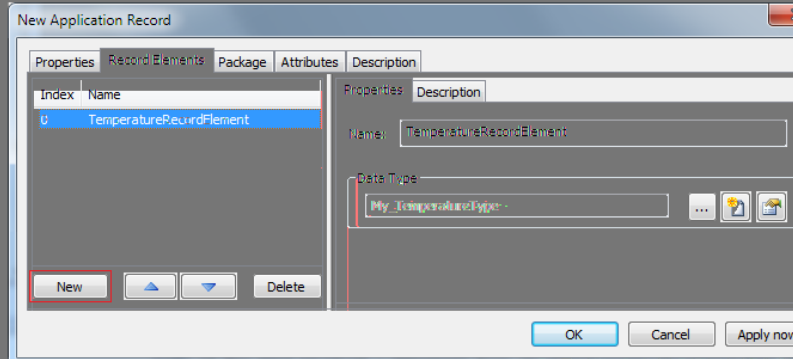
1. Enter the name of the record type



2. Click "New" in the "Record Elements" tab



data type 3. Enter the name "TemperatureRecordElement" and set the reference to "My_TemperatureType"



4. Create a second record element.

Enter the name "SpeedRecordElement" and set the reference to "My_SpeedType" data type

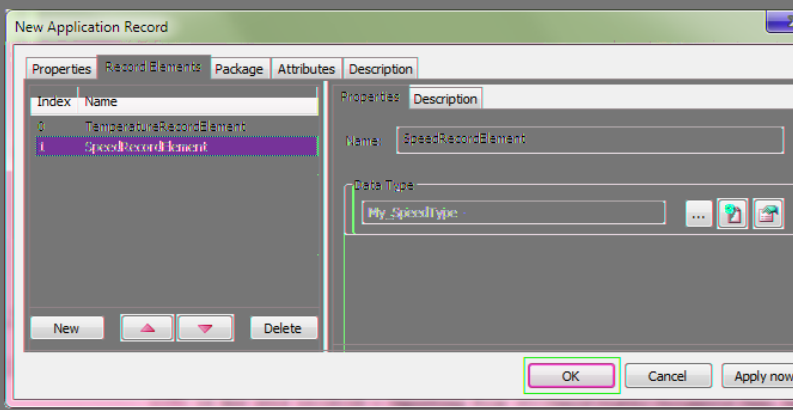


Figure 4-19 Record type creation with 2 record elements

- The corresponding implementation data type is created:

The corresponding implementation data type shall have a structure corresponding to the application data type so that each sub element of the application level has a corresponding element at implementation level. In this example a record implementation data type with 2 sub elements is used.

- Right click on “Implementation Data Type” library is done and “New Record...” is selected (see Figure 4-20).
- The record data type is configured (see Figure 4-21):
- The name is set: “My_RecordImplementationType”
- In the “Record Elements” tab, a record element matching with temperature at implementation level is created:

The name “TemperatureImplementationRecordElement” is set.

The element type is set to “Type Reference” in order not to create a new implementation data type, but to use an existing one.

The property button is clicked to open the dialog where this reference is set. The type sint16 is chosen corresponding to the implementation data type mapped to “My_TemperatureType”.

- In a similar way a record matching with speed at implementation level is created:

The name “SpeedImplementationRecordElement” is set.

The element type is set to “Type Reference” in order not to create a new implementation data type, but to use an existing one.

The property button is clicked to open the dialog where this reference is set. The type uint16 is chosen corresponding to the implementation data type mapped to “My_SpeedType”.

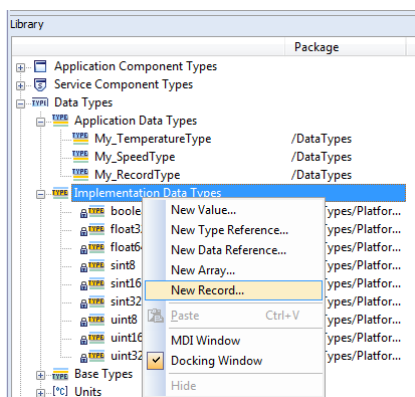
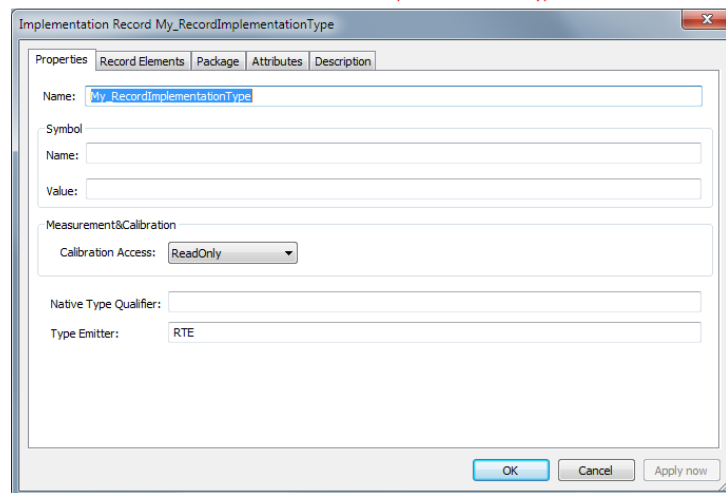
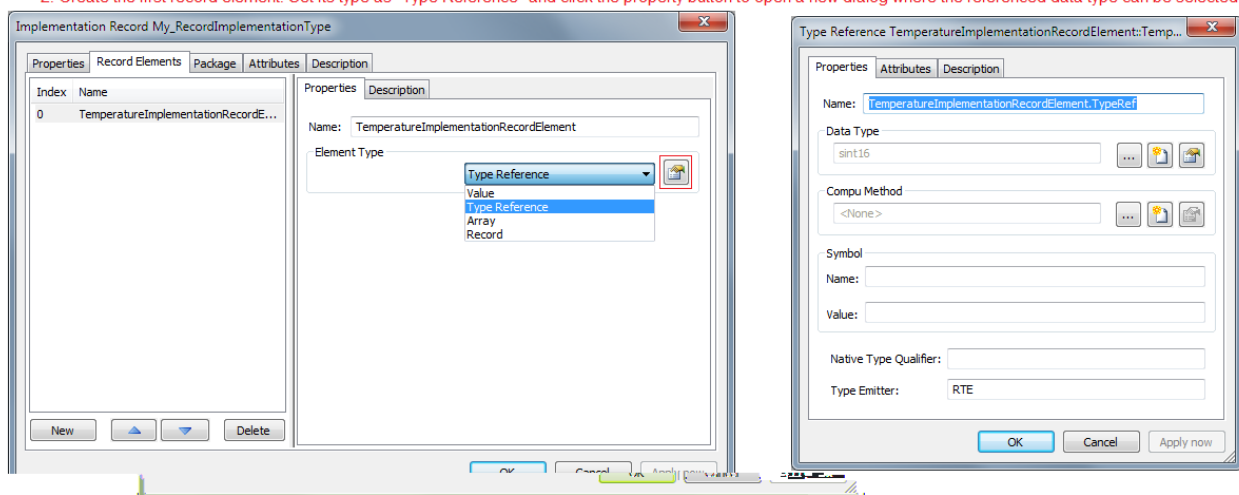


Figure 4-20 New Record implementation data type

1. Set the name of the record implementation data type



2. Create the first record element. Set its type as "Type Reference" and click the property button to open a new dialog where the referenced data type can be selected



3. Repeat the operation for the next record elements

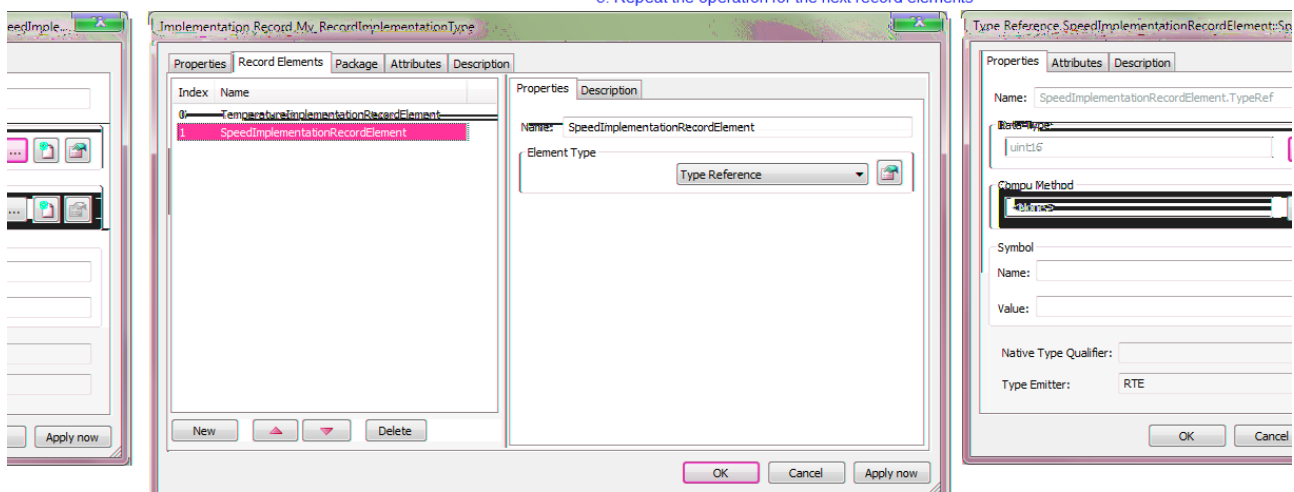


Figure 4-21 Set a record implementation data type

- The record application data type is now mapped with the record implementation data type (see Figure 4-22).



Figure 4-22 Record type mapping



Note

When the 2 record types are mapped, the “only show compatible Implementation Data Types” is very useful for 2 reasons:

- It provides a reduced list of available types, which makes the selection easier
 - It already gives the information if the 2 records types are compatible or not. In case the desired implementation data type does not appear in the selection list, then the designer has already the information that the types do not match and that the configuration is not correct.
-

- A sender/receiver port interface is created, where the data element is named „RecordElement” and typed by „My_RecordType“ (see Figure 4-23).

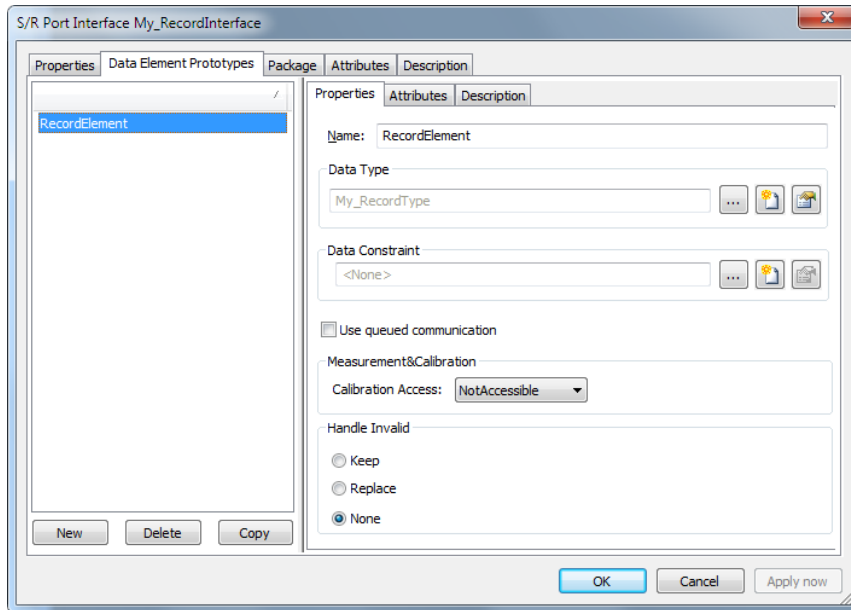


Figure 4-23 Sender/receiver port interface with record element

- A P-Port prototype referencing „My_RecordInterface” is created in SWC_Sender and accessed by „SWC_SenderRunnable”. A similar R-Port is created at SWC_Receiver and accessed by “SWC_ReceiverRunnable”.

For both ports an init value shall be set in the com spec. Two ways can be followed to set the init values:

- > Manually at the port (see Figure 4-24 and Figure 4-25).
- > With a constant: a constant is created (see Figure 4-26) and this constant is referenced as init value (see Figure 4-27).

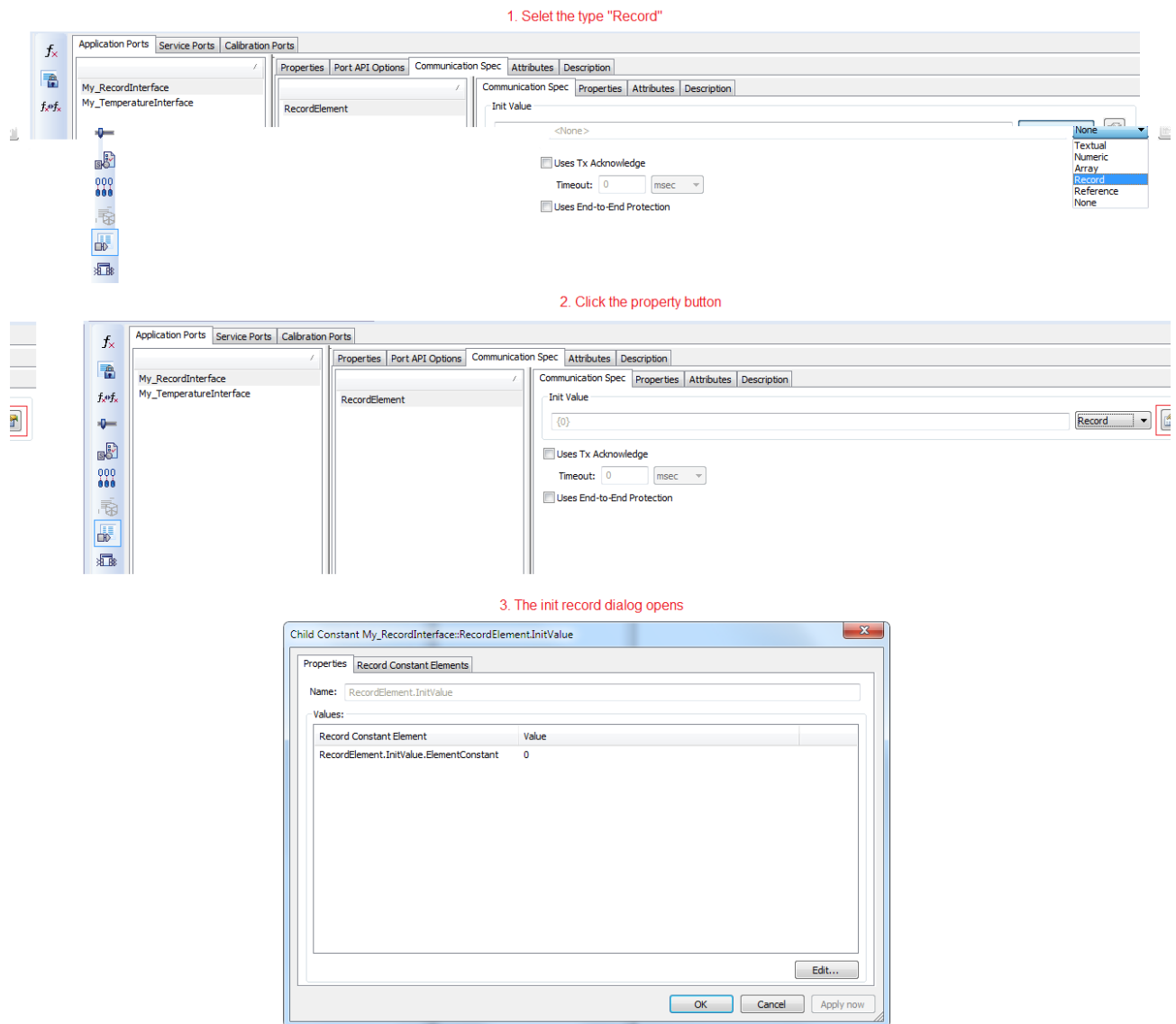
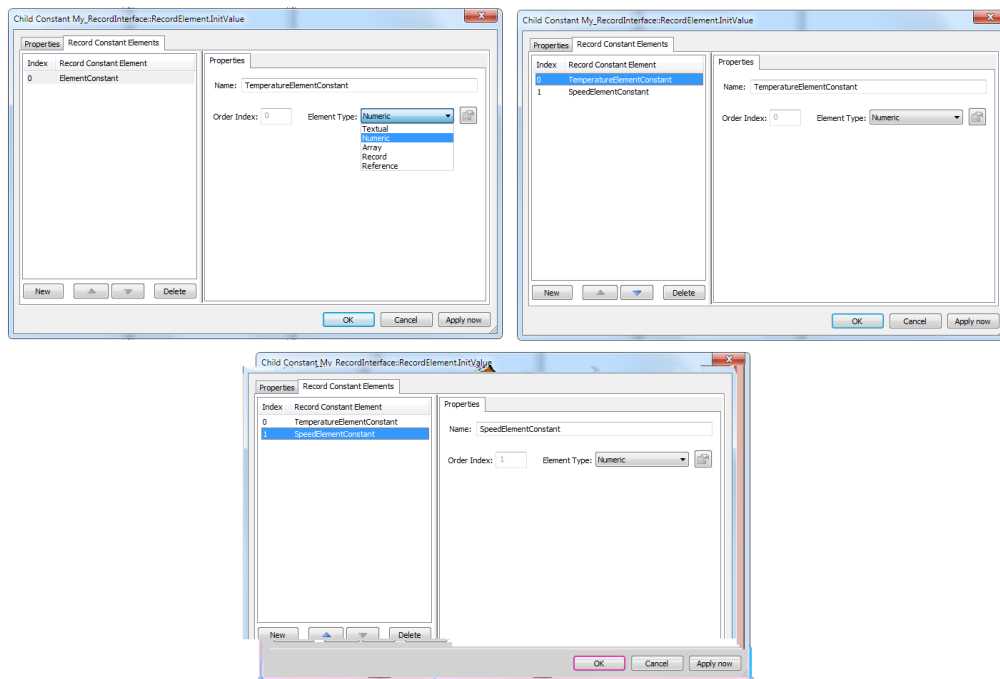
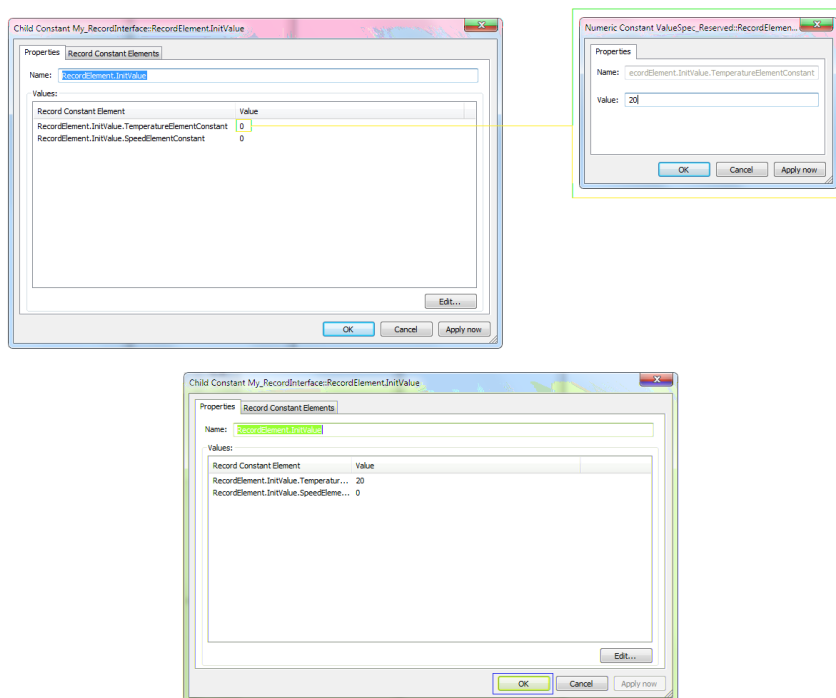


Figure 4-24 Record manual init – part 1

3. In the "Record Constant Elements" tab create the record element constants as "Numeric".



4. In the "Properties" tab set the single values of the record elements by double clicking on the lines.



5. After clicking "Ok" the init record value is visible in the port prototype editor.

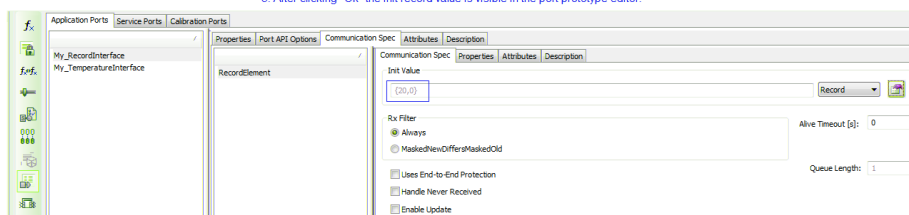
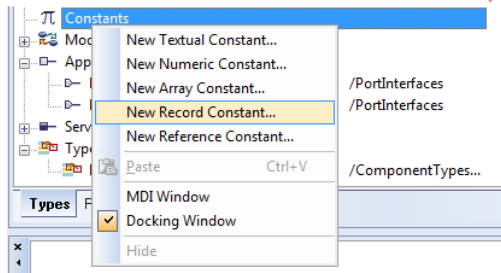
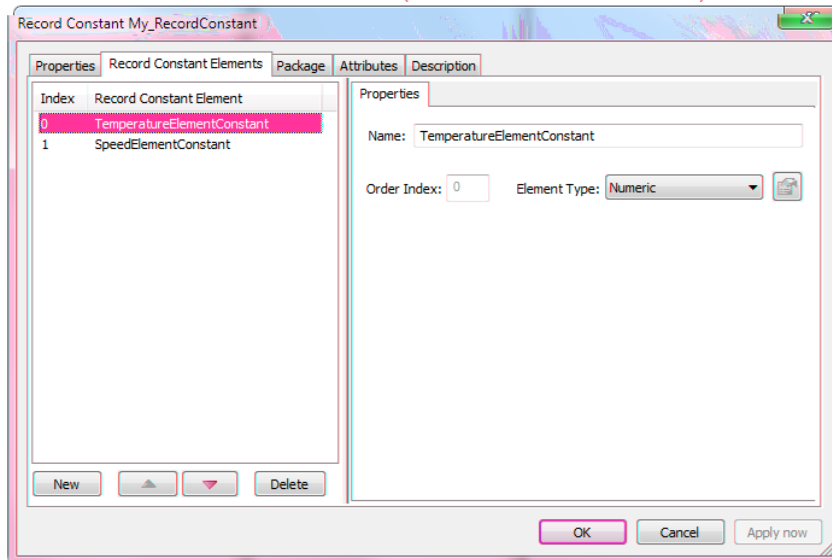


Figure 4-25 Record manual init – part 2

1. Create a new Record constant in the "Constants" library



2. Create the record elements (similar to manual init value creation)



3. Set the value for each record element by double-clicking on the corresponding line

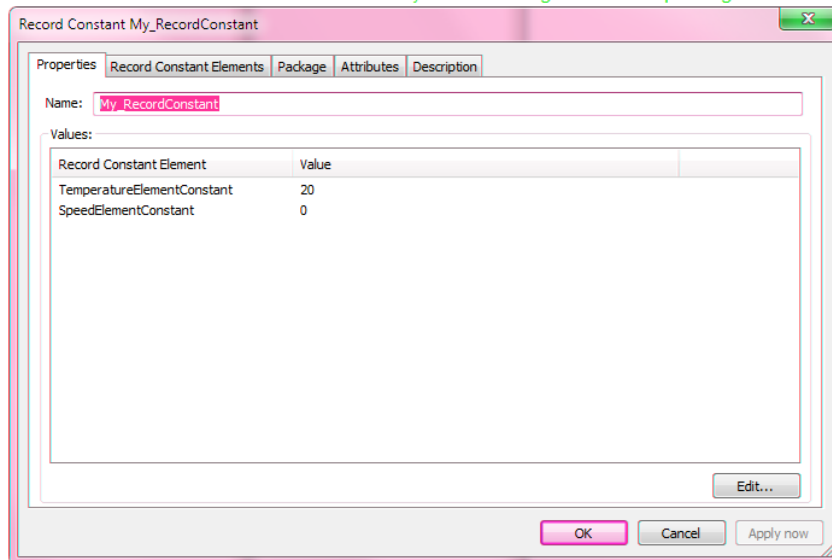


Figure 4-26 Record constant creation

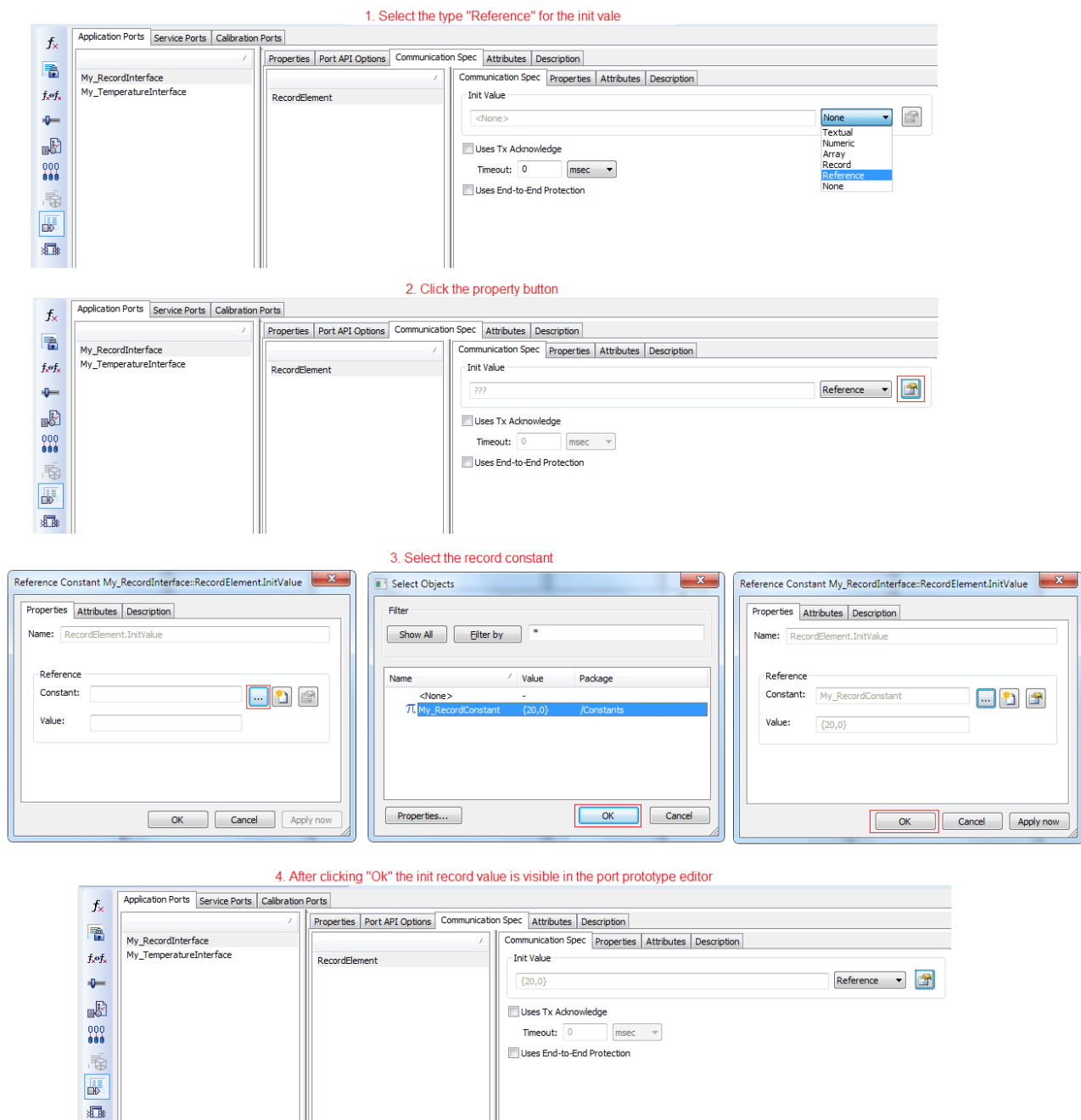


Figure 4-27 Referencing a record constant as init value

The 2 ports are then connected (see Figure 4-28).

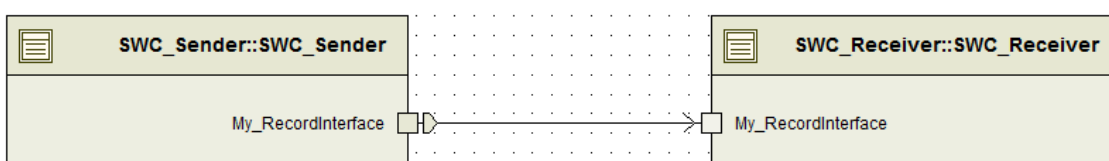


Figure 4-28 My_RecordInterface port connection

- ▶ “My_TypeMappingSet” has to be referenced by SWC_Sender and SWC_Receiver, which was already done in chapter 4.1.
- ▶ The system description resulting from the configuration done in DaVinci Developer is imported in DaVinci Configurator Pro. There the Rte generation and the SWC generation are performed.

The following code is generated:

In SWC_Sender.c:

```
Std_ReturnType Rte_Write_ (const *data)
```

In SWC_Receiver.c:

```
Std_ReturnType Rte_Read_ ( *data)
```

In Rte_Type.h:

```
# define Rte_TypeDef_
typedef struct
{
    sint16
    uint16
};
```

In Rte.c:

```
CONST ( , RTE_CONST) Rte_C_ _0 = {
    40, 0U
};

CONST ( , RTE_CONST) Rte_ = {
    40, 0U
};
```

The record data element is typed by My_RecordImplementationType.

The record type has 2 elements, TemperatureImplementationRecordElement of type sint16 and SpeedImplementationRecordElement of type uint16.

Two constants are generated for the init values: Rte_My_RecordConstant corresponding to the constant created in DaVinci Developer and Rte_C_MyRecordImplementationType_0 corresponding to the init value set manually.

4.3 Enumeration

Two enumeration examples are described in this chapter. The first one (see 4.3.1) corresponds to enumeration created at application level. The second one (see 0) corresponds to enumeration created at implementation level.

4.3.1 Application level

In this example an enumeration application data type is created. It defines 3 text values and is mapped to uint8.

- ▶ A new Value application data type is created: “My_EnumerationType” (see Figure 4-29):
 - ▶ It references “My_EnumerationType_CompuMethod” which defines a text table conversion (see Figure 4-30).
 - ▶ This text table contains 3 values (see Figure 4-31):
 - Text_00 corresponding to numerical value 0 (range [0;0])
 - Text_01 corresponding to numerical value 1 (range [1;1])
 - Text_02 corresponding to numerical value 2 (range [2;2])
- ▶ “My_EnumerationType” references “My_EnumerationType_Constraint” which specifies [0;2] as physical range (see Figure 4-32).
- ▶ „My_EnumerationType is then mapped to uint8 in „My_TypeMappingSet” (see Figure 4-33).

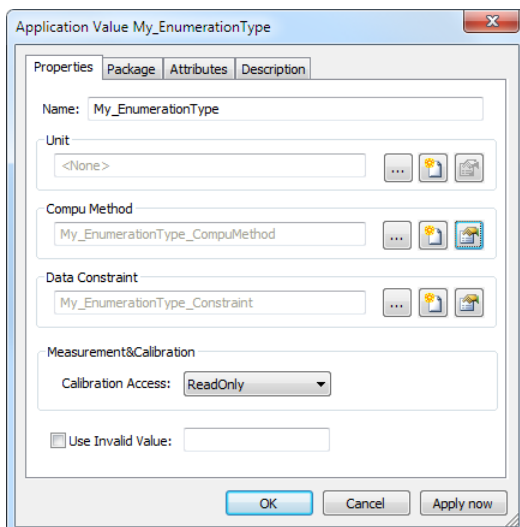


Figure 4-29 My_Enumeration type

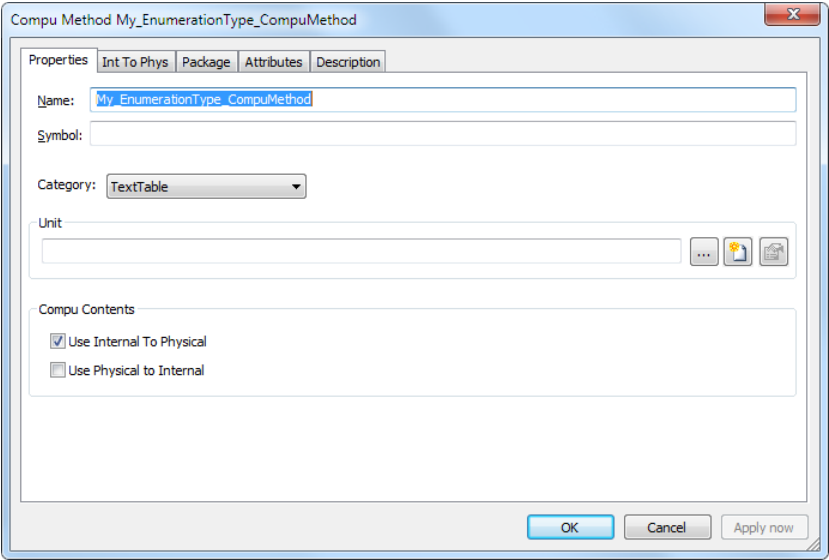


Figure 4-30 My_EnumerationType_CompuMethod

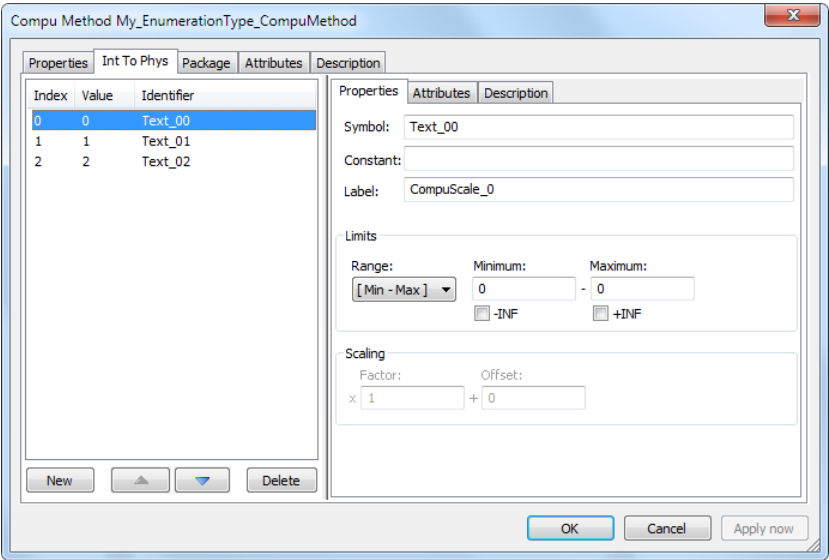
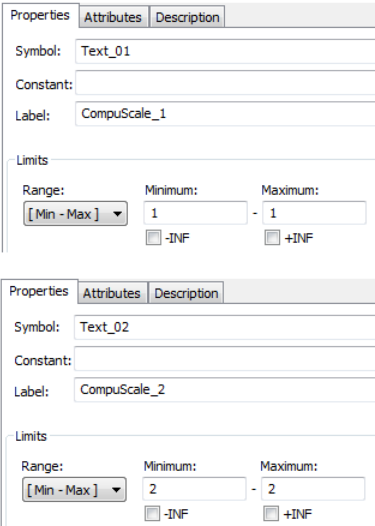


Figure 4-31 Enumeration text table setting



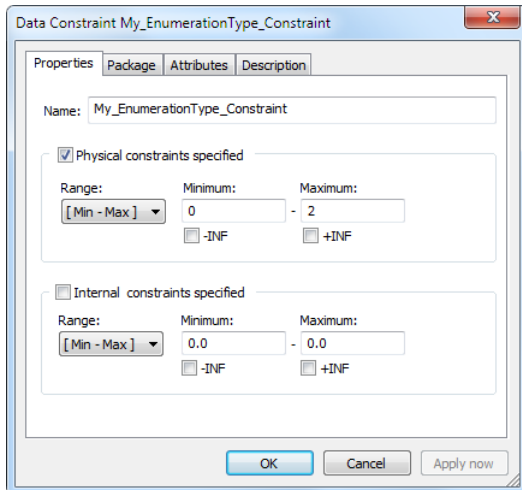


Figure 4-32 Enumeration constraint

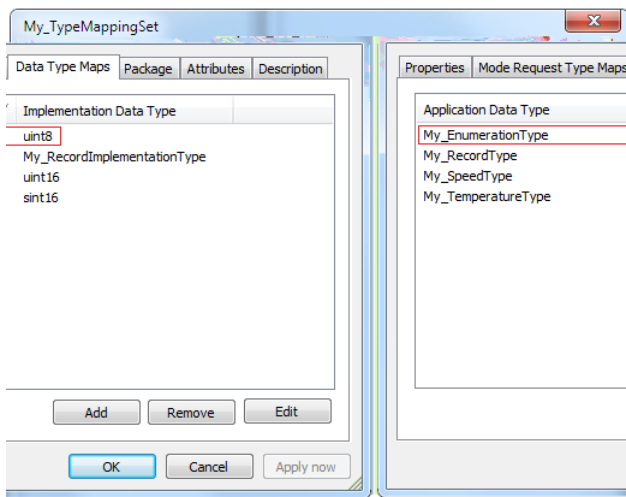


Figure 4-33 My_Enumeration mapping to uint8

- A sender/receiver port interface “My_EnumerationInterface” is created. It contains one data element of type “My_EnumerationType” (see Figure 4-34).

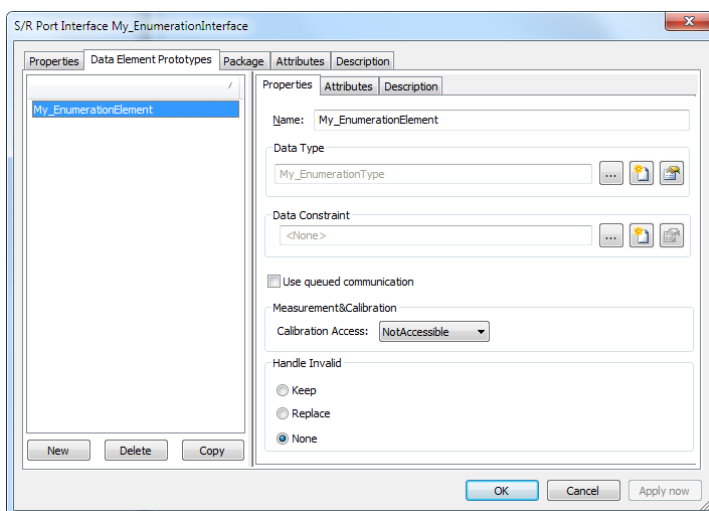


Figure 4-34 “My_EnumerationInterface”

- ▶ A P-Port is created on the SWC_Sender with write access to the data element in the SWC_SenderRunnable, a R-Port is created on the SWC_Receiver with read access to the data element in the SWC_ReceiverRunnable, init values are specified and both ports are connected.
- ▶ “My_TypeMappingSet” has to be referenced by SWC_Sender and SWC_Receiver, which was already done in chapter 4.1.
- ▶ The system description resulting from the configuration done in DaVinci Developer is imported in DaVinci Configurator Pro. There the Rte generation and the SWC generation are performed.

The following code is generated:

In SWC_Sender.c:

```
Std_ReturnType Rte_Write_          _          (uint8 data)
```

In SWC_Receiver.c:

```
Std_ReturnType Rte_Read_          _          (uint8 *data)
```

In Rte_SWC_Sender_Type.h and Rte_SWC_Receiver_Type.h:

```
# define          _LowerLimit (0U)
# define          _UpperLimit (2U)

# ifndef Text_00
#   define          (0U)
# endif

# ifndef Text_01
#   define          (1U)
# endif

# ifndef Text_02
#   define          (2U)
# endif
```

The data element is typed by uint8.

The lower limit is 0 and upper limit is 2.

Each enumeration text is defined with the corresponding uint8 value

4.3.2 Implementation level

In this example an enumeration implementation data type is created. It defines 3 text values and has uint8 as reference type.

- ▶ A new Reference implementation data type is created: “My_EnumerationImplType” (see Figure 4-35):
- ▶ It references “My_EnumerationImplType_CompuMethod” which defines a text table conversion (see Figure 4-36).
- ▶ This text table contains 3 values, Text_00 for value 0 (range [0;0]), Text_01 for value 1 (range [1;1]) and Text_02 for value 2 (range [2;2]) (see Figure 4-37).

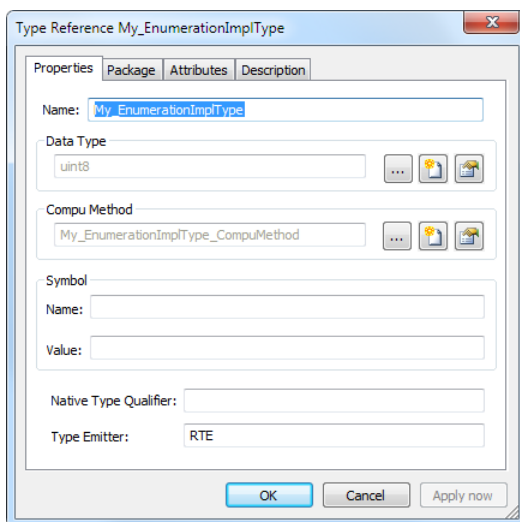


Figure 4-35 My_EnumerationImplType

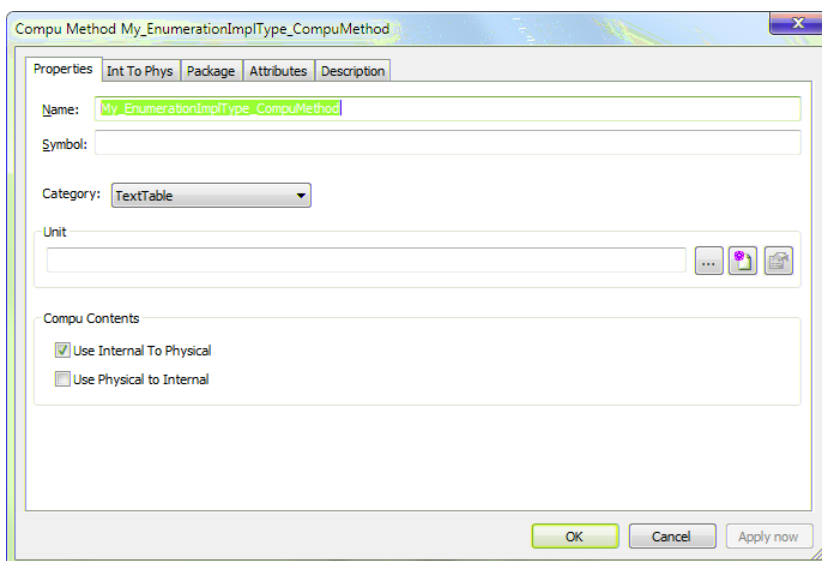


Figure 4-36 My_EnumerationImplType_CompuMethod

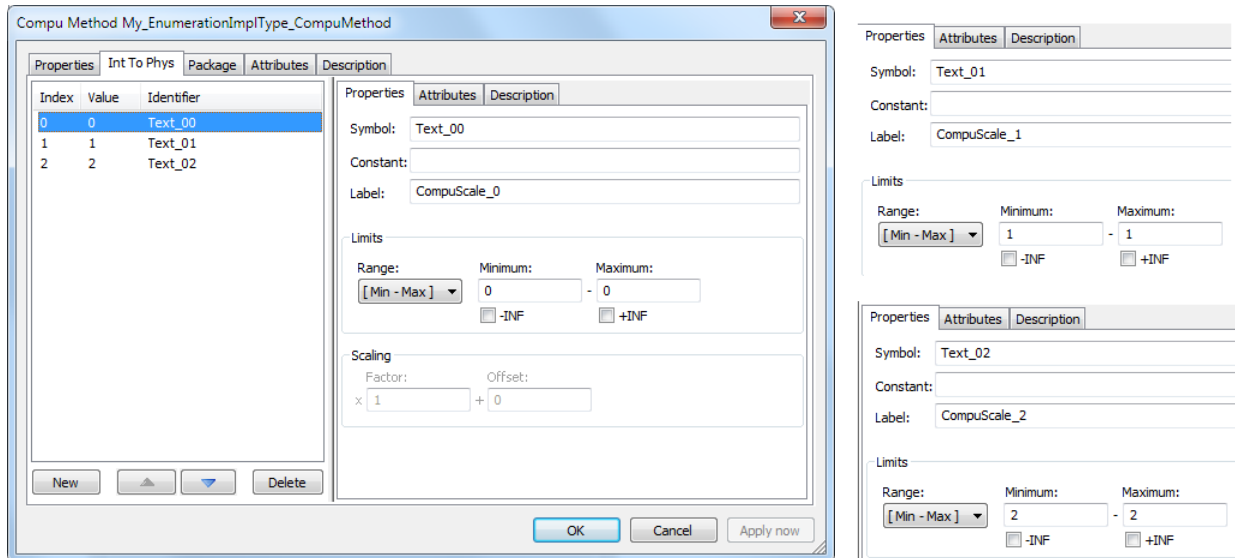


Figure 4-37 My_EnumerationImplType_CompuMethod text table

- ▶ A sender/receiver port interface “My_EnumerationImplInterface” is created. It contains one data element of type “My_EnumerationImplType” (see Figure 4-38).

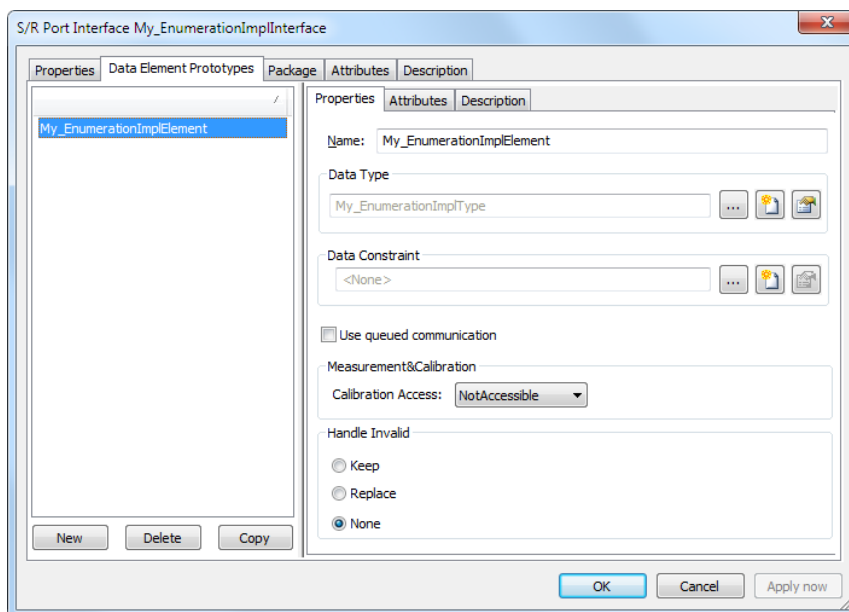


Figure 4-38 My_EnumerationImplInterface

- ▶ A P-Port is created on the SWC_Sender with write access to the data element in the SWC_SenderRunnable, an R-Port is created on the SWC_Receiver with read access to the data element in the SWC_ReceiverRunnable, init values are specified and both ports are connected.
- ▶ The system description resulting from the configuration done in DaVinci Developer is imported in DaVinci Configurator Pro. There the Rte generation and the SWC generation are performed.

The following code is generated:

In SWC_Sender.c:

```
Std_ReturnType
Rte_Write_          _          (          data)

* Enumeration Types:
* =====
* My_EnumerationImplType: Enumeration of integer in interval [0...255] with enumerators
*          (0U)
*          (1U)
*          (2U)
```

In SWC_Receiver.c:

```
Std_ReturnType Rte_Read_          _          (
*data)

* Enumeration Types:
* =====
* My_EnumerationImplType: Enumeration of integer in interval [0...255] with enumerators
*          (0U)
*          (1U)
*          (2U)
```

In Rte_SWC_Sender_Type.h and Rte_SWC_Receiver_Type.h:

```
# define          _LowerLimit (0U)
# define          _UpperLimit (2U)

# ifndef Text_00
#   define          (0U)
# endif

# ifndef Text_01
#   define          (1U)
# endif

# ifndef Text_02
#   define          (2U)
# endif
```

In Rte_Type.h:

```
# define Rte_TypeDef_
typedef uint8          ;
```

The data element is typed by My_EnumerationImplType which is defined as uint8.
Each enumeration text is defined with the corresponding uint8 value

4.4 Mode declaration

A mode transmitted via mode switch port interfaces is a data element that can take its values in a corresponding mode declaration group.

Mode declaration groups can be created in two different ways. The first one is to create a mode declaration group manually in DaVinci Developer, which will then be used in a mode switch port interface by SWCs and/or by BswM (see 4.4.1). The second one is to create a mode service port in the BswM configuration in DaVinci Configurator Pro, which will automatically create a corresponding mode declaration group (see 4.4.2).

4.4.1 Mode declaration group in DaVinci Developer

- It is possible to create mode declaration groups in DaVinci Developer by right-clicking “Mode Declaration Group” library and selecting “New Mode Declaration Group...” (see Figure 4-39).

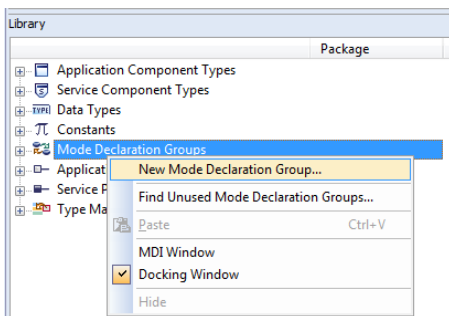


Figure 4-39 New mode declaration group

- The mode declaration group is configured (see Figure 4-40):
 - The name is given
 - The category is set: alphabetic or explicit. Alphabetic means that the mode declaration internal values will be attributed automatically in the alphabetical order. Explicit means that the designer can set these values himself.
 - The mode declarations are entered.

The internal values of mode declarations and transition may be entered, but are not mandatory (see the note below).



Note

The Rte generator supports currently only alphabetic category. That means, even if “Explicit” is set and values are attributed by the designer to the mode declarations, the mode declaration internal values are automatically generated following the alphabetical order.

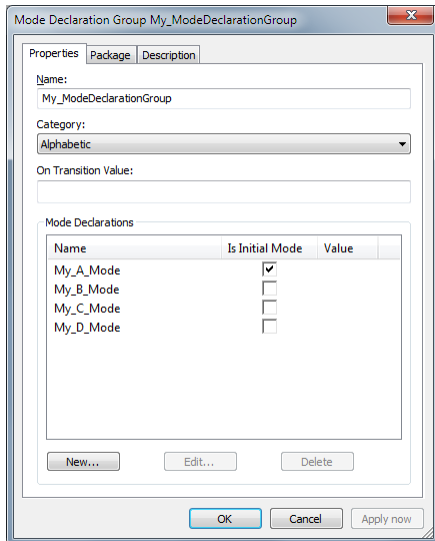


Figure 4-40 My_ModeDeclarationGroup

- A mode port interface called “My_ModePortInterface” is created. It transmits “My_Mode” of mode declaration group “My_ModeDeclarationGroup” (see Figure 4-41).

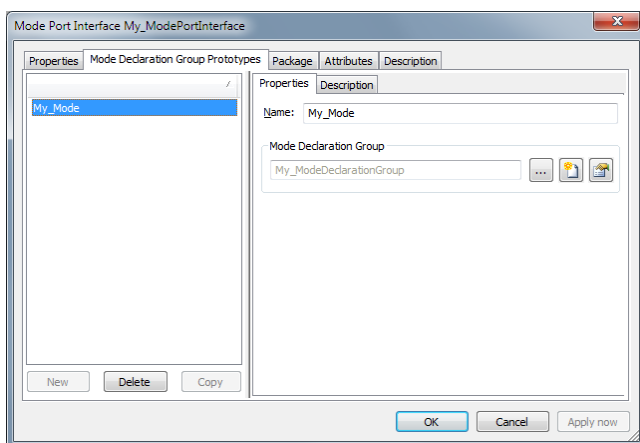


Figure 4-41 My ModePortInterface

- In this example SWC_Sender is the mode master and SWC_Receiver is the mode user.

SWC_Sender resp. SWC_Receiver implements a sender resp. sender port prototype referencing “My_ModePortInterface”. They are connected to each other (see Figure 4-42). SWC_SenderRunnable has “Send Mode Switches” port access on the mode switch port, SWC_ReceiverRunnable has “Read Mode” port access on the mode switch port (see Figure 4-43).



Figure 4-42 Mode ports connection

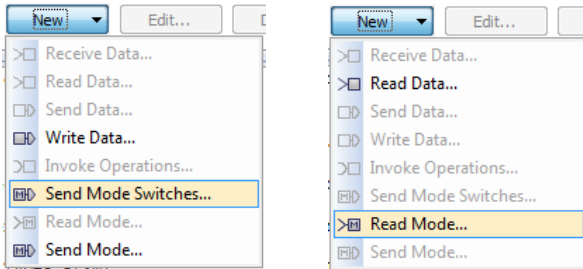


Figure 4-43 Mode port access



Note

The type for the mode declaration group in the code is automatically the corresponding platform type: if the number of mode declarations is lower than 256, then the mode declaration group is mapped automatically to uint8. If this number is bigger than 256 and lower than 65536, then it is mapped to uint16, and so on...

The system description resulting from the configuration done in DaVinci Developer is imported in DaVinci Configurator Pro. There the Rte generation and the SWC generation are performed.

The following code is generated:

In SWC_Sender.c:

```
* Mode Interfaces:
* =====
*
*                                     Std_ReturnType
Rte_Switch_My_ModePortInterface_My_Mode(Rte_ModeType_My_ModeDeclarationGroup
mode)
*   Modes of Rte_ModeType_My_ModeDeclarationGroup:
*   - RTE_MODE_My_ModeDeclarationGroup_My_A_Mode
*   - RTE_MODE_My_ModeDeclarationGroup_My_B_Mode
*   - RTE_MODE_My_ModeDeclarationGroup_My_C_Mode
*   - RTE_MODE_My_ModeDeclarationGroup_My_D_Mode
*   - RTE_TRANSITION_My_ModeDeclarationGroup
```

In SWC_Receiver.c:

```
* Mode Interfaces:
* =====
*
*                                     Rte_ModeType_My_ModeDeclarationGroup
Rte_Mode_My_ModePortInterface_My_Mode(void)
*   Modes of Rte_ModeType_My_ModeDeclarationGroup:
*   - RTE_MODE_My_ModeDeclarationGroup_My_A_Mode
*   - RTE_MODE_My_ModeDeclarationGroup_My_B_Mode
*   - RTE_MODE_My_ModeDeclarationGroup_My_C_Mode
*   - RTE_MODE_My_ModeDeclarationGroup_My_D_Mode
*   - RTE_TRANSITION_My_ModeDeclarationGroup
*
```

In Rte_Type.h:

```
typedef uint8 Rte_ModeType_My_ModeDeclarationGroup;

# define RTE_MODE_My_ModeDeclarationGroup_My_A_Mode (0U)
# define RTE_MODE_My_ModeDeclarationGroup_My_B_Mode (1U)
# define RTE_MODE_My_ModeDeclarationGroup_My_C_Mode (2U)
# define RTE_MODE_My_ModeDeclarationGroup_My_D_Mode (3U)
# define RTE_TRANSITION_My_ModeDeclarationGroup (4U)
```

The mode declaration group is typed by uint8.

The mode declaration values are attributed in the alphabetical order.

4.4.2 Mode service port in BswM configuration

In DaVinci Configurator Pro, the BswM BSWMD offers the possibility to create mode switch ports referencing pre-defined mode declaration groups.

For instance it is possible to create a BswM switch port referencing the ComM mode declaration group (see Figure 4-44).

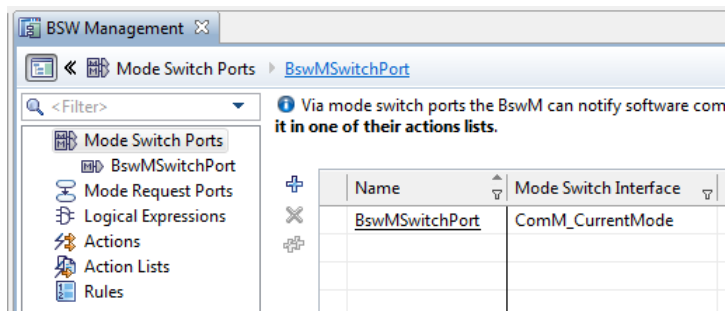


Figure 4-44 BswM configuration

After configuring the BswM in DaVinci Configurator Pro, it is possible to import its service component description in DaVinci Developer. As shown Figure 4-45 BswM is imported in the “Service Component Types” library, the mode switch interface referencing the ComM mode in the “Service Port Interfaces” library.

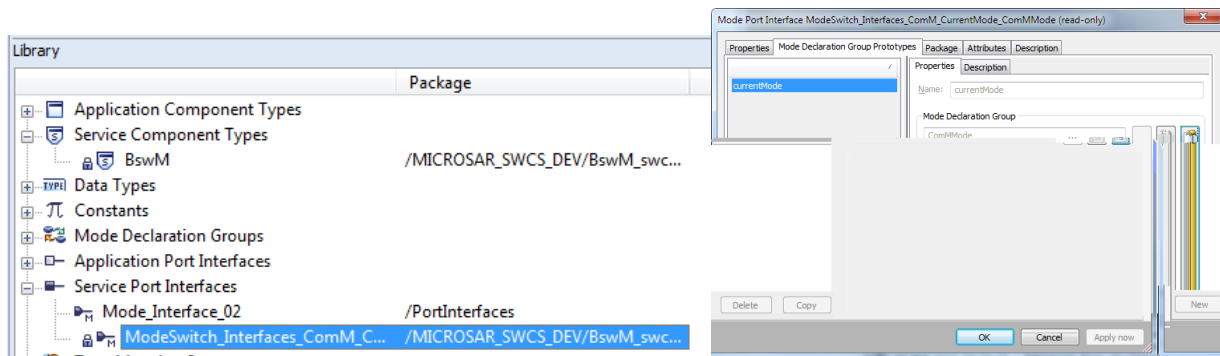


Figure 4-45 BswM import in DaVinci Developer

The mode declaration group ComMMode is also imported in the “Mode Declaration Group” library (see Figure 4-46) and the corresponding mode declarations are visible by opening it (see Figure 4-47).

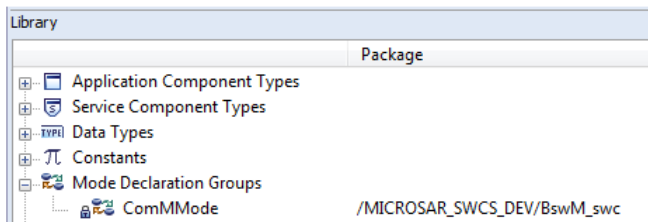


Figure 4-46 Mode declaration group import in DaVinci Developer

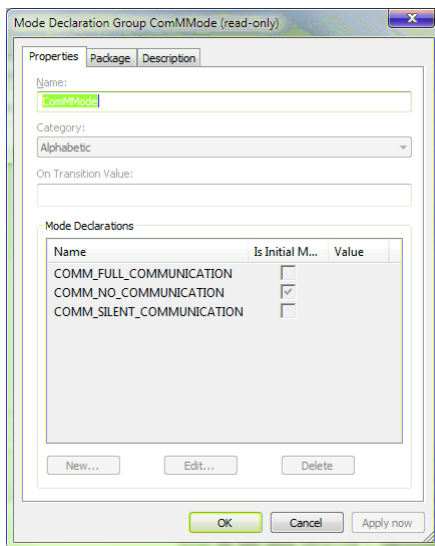


Figure 4-47 ComM mode declaration group

This configuration gives then the following generated code:

In SWC_A.c:

```
Rte_ModeType_ComMMode
Rte_Mode_ModeSwitch_Interfaces_ _ _currentMode(void)

//Modes of Rte_ModeType_ :
// - RTE_MODE_ComMMode_COMM_FULL_COMMUNICATION
// - RTE_MODE_ComMMode_COMM_NO_COMMUNICATION
// - RTE_MODE_ComMMode_COMM_SILENT_COMMUNICATION
// - RTE_TRANSITION_ComMMode
```

In Rte_SWC_A.h:

```
//Buffers for Mode Management
extern VAR(Rte_ModeType_ComMMode, RTE_VAR_NOINIT)
Rte_ModeMachine_BswM_ModeSwitch_Interfaces_ _ _currentMode
;
```

In Rte.c:

```
/* mode management initialization part 1 */
Rte_ModeMachine_BswM_ModeSwitch_Interfaces_ _ComMMode_currentMode
= RTE_MODE_ _COMM_NO_COMMUNICATION;
```

In Rte_Type.h:

```
//define Rte_TypeDef_ComM_ModeType
typedef uint8          ;

//Definitions for Mode Management
typedef uint8 Rte_ModeType_          ;

# define RTE_MODE_ComMMode_COMM_FULL_COMMUNICATION (0U)
# define RTE_MODE_ComMMode_COMM_NO_COMMUNICATION (1U)
# define RTE_MODE_ComMMode_COMM_SILENT_COMMUNICATION (2U)
# define RTE_TRANSITION_ComMMode (3U)
```

The mode declaration group Com_ModeType is typed by uint8.

The mode declarations are available as defines and their uint8 value is given in the alphabetical order.

4.4.3 Mode request port and mapping

A component requesting a mode change to the mode master shall implement a sender/receiver P-Port as mode request port. The corresponding port interface's data element shall be typed by an implementation data type referencing a compu method containing an enumeration of the corresponding modes. That implementation data type shall be mapped to the corresponding mode declaration group.

- The mode declaration group for the example will be `My_ModeDeclarationGroup` presented Figure 4-48.

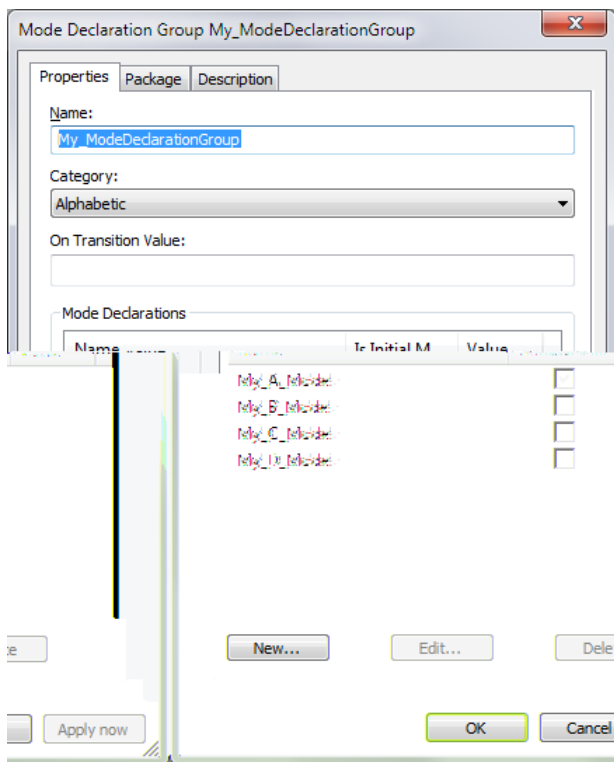


Figure 4-48 My_ModeDeclarationGroup

- A corresponding implementation data type is created: `My_ModeDeclarationGroup_IType`.
 - It is defined as type reference to `uint8`
 - It contains a compu method which defined an enumeration of the 4 modes from `My_ModeDeclarationGroup`. The same strings shall be used. See Figure 4-49.

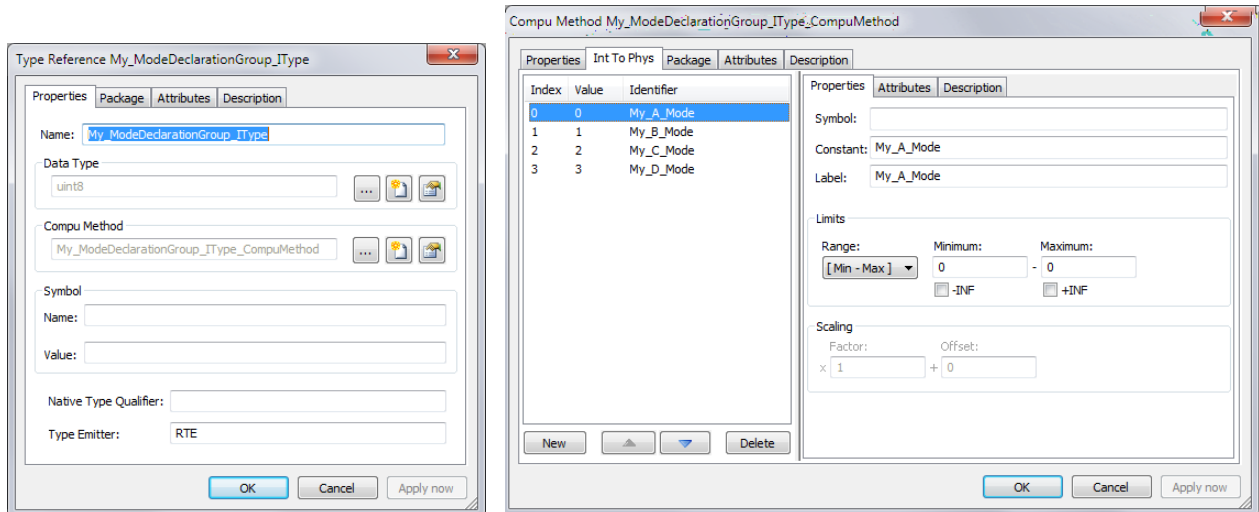


Figure 4-49 My_ModeDeclarationGroup_IType

- The mode declaration group and the implementation data type are mapped in a ModeRequestTypeMap (Figure 4-50).

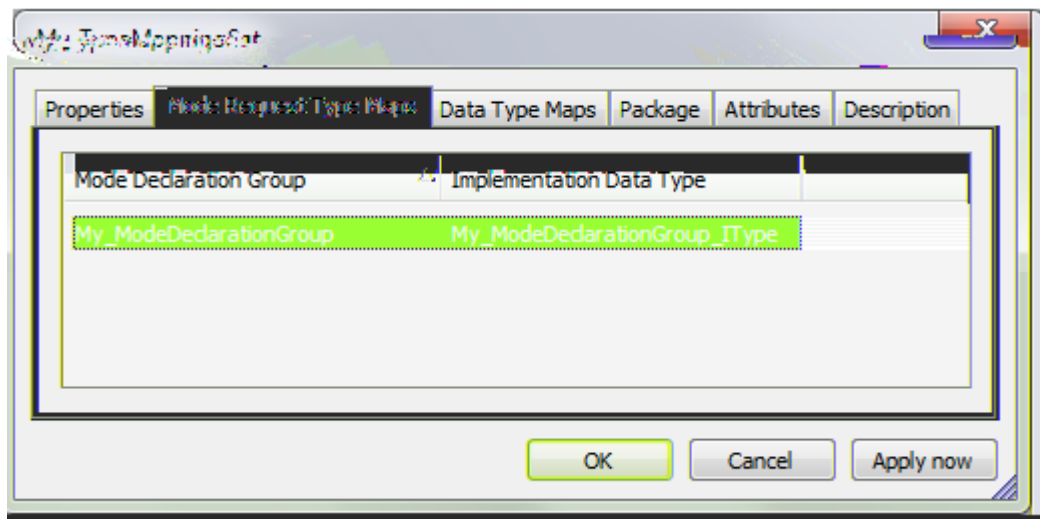


Figure 4-50 Mode declaration group mapping

- The corresponding SWC shall implement a port and reference the type mapping set.
- A service sender/receiver port interface is created containing 1 data element typed by My_ModeDeclarationGroup_IType (Figure 4-51).

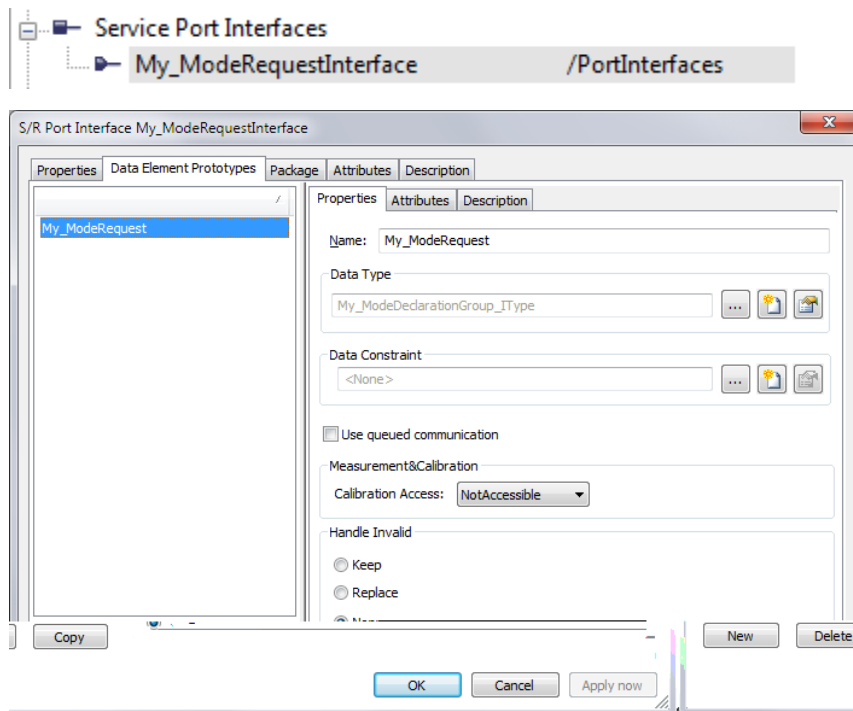


Figure 4-51 My_ModeRequestInterface

- A service P-Port prototype referencing this port interface is created on the SWC (Figure 4-52).

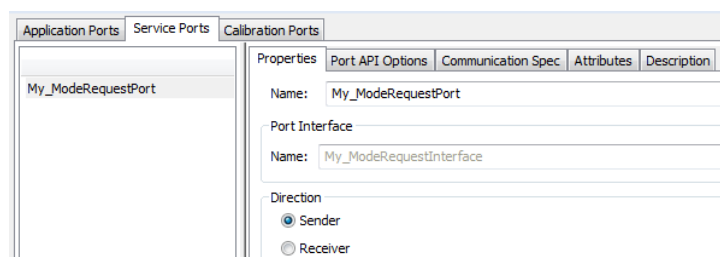


Figure 4-52 My_ModeRequestPort

- The type mapping set is referenced by the SWC (Figure 4-53).

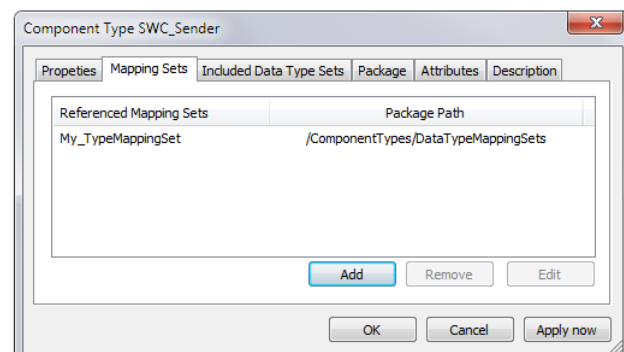


Figure 4-53 Type mapping set reference for mode request

- ▶ A runnable of the SWC shall have write access on the created port.
- ▶ In DaVinci Configurator, the corresponding mode request port on BswM can be created and mapped to the SWC port.
- ▶ The generated code provides the following for the SWC:

Type and enumeration definition:

```
* Enumeration Types:
* =====
* My_ModeDeclarationGroup_IType: Enumeration of integer in interval [0...255] with
enumerators
*   My_A_Mode (0U)
*   My_B_Mode (1U)
*   My_C_Mode (2U)
*   My_D_Mode (3U)
```

Mode request API for the SWC:

```
Std_ReturnType Rte_Write_My_ModeRequestPort_My_ModeRequest(My_ModeDeclarationGroup_IType
data)
```

4.5 Implementation data type examples

As a summary, the two following examples are showing how implementation data types can be created as type reference (see 4.5.1) or as value (see 0).

4.5.1 Type reference

It is possible to create an implementation data type out of an existing implementation data type. This is done by right-clicking the “Implementation Data Types” library and selecting “New Type Reference...”.

As shown Figure 4-54, a name is given, as well as a reference to another implementation data type in the “Data Type” field. For the example uint8 is chosen.

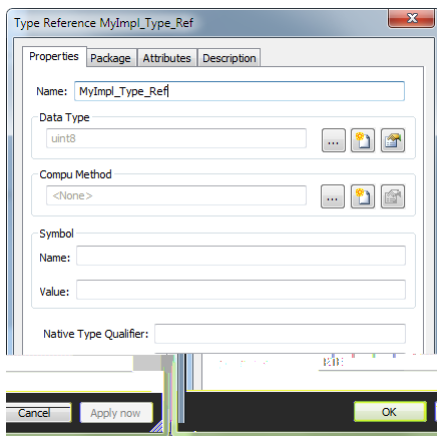


Figure 4-54 New type reference

At Rte generation, the following is generated in Rte_Type.h:

```
# define Rte_TypeDef_MyImpl_Type_Ref
typedef uint8
```

The new created implementation data type is used and typed via typedef by the referenced data type.

4.5.2 Value

- ▶ It is possible to create an implementation data type of category value referencing a base type. This is done in this example by right-clicking the “Implementation Data Types” library and selecting “New Value...”.
- ▶ The Value implementation data type gets a name: “My_ImplValue” in this example.
- ▶ A reference to a base type has to be given in the “Base Type” field (see Figure 4-55). In this example a “My_ImplValue_base” base type is used (see Figure 4-56). A valid “Native Declaration” must be set in the base type: unsigned char here.
- ▶ It is also optionally possible to set a constraint to the implementation data type, which is done in My_ImplValue_Constraint (see Figure 4-57). It is set here that the internal values are limited to the range [0;150].

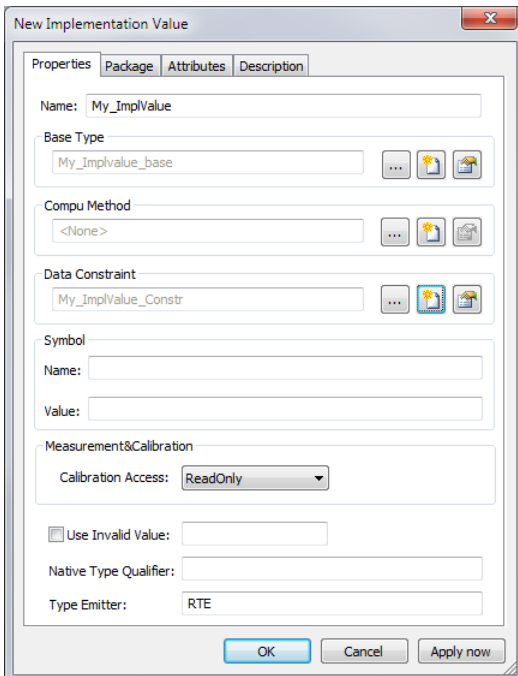


Figure 4-55 Value implementation data type

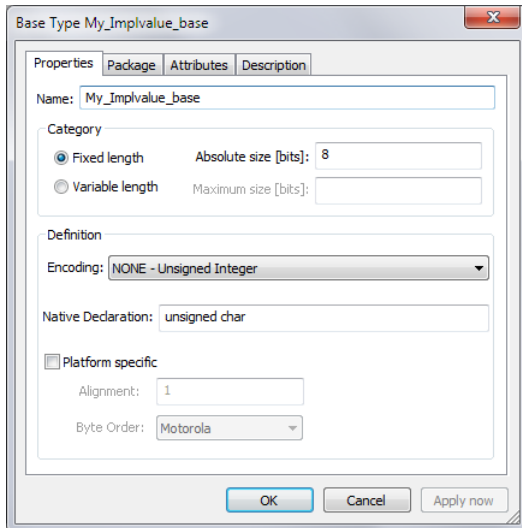


Figure 4-56 My_ImplValue_base base type

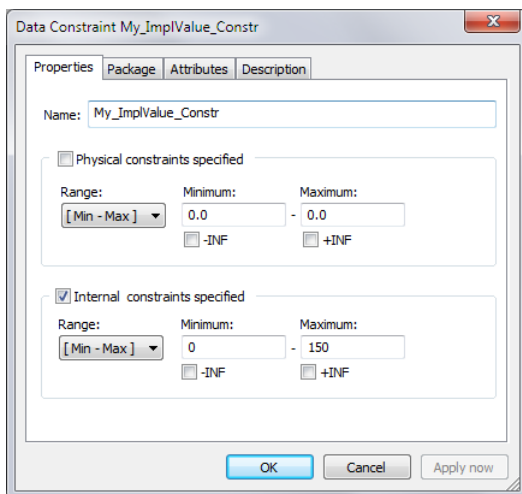


Figure 4-57 Implementation type constraint

- The configured implementation data type is generated via Rte generator and the following can be seen:

In Rte_Type.h:

```
# define Rte_TypeDef_My_ImplValue
typedef unsigned char My_ImplValue;

//Primitive Types:
// =====
//My_ImplValue: Integer in interval [0...150]
```

The new created implementation data type is typed via typedef by the native declaration of the referenced base type. As explained chapter 3.5, this example introduces a platform dependency because “unsigned char” meaning is platform dependent.

The constraint set in the configuration appears only as comments in the code.

5 Additional information

5.1 Compatibility and conversion

- It is possible to create several implementation data types with the same name if they belong to 2 different packages.

At code generation only 1 implementation data type is generated. In case the implementation data types with the same name are not exactly the same (differences in compu methods, constraints...) an information message #40286 is displayed in DaVinci Developer and in DaVinci Configurator Pro (see Figure 5-1).

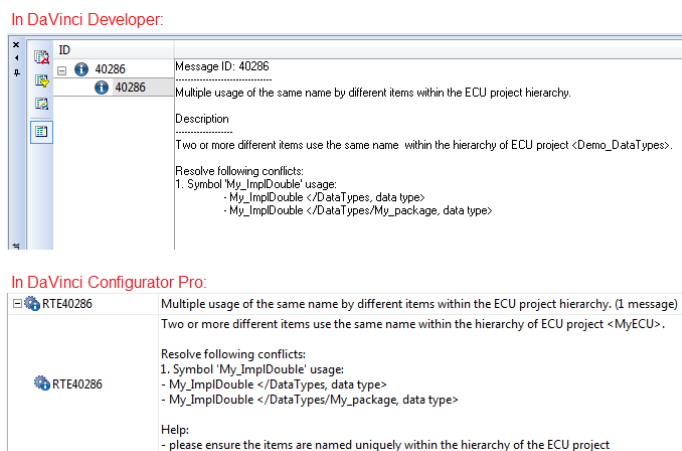


Figure 5-1 Info Message #40286

- Compatibility of ports is verified in DaVinci Developer based on the name and the category of the connected data elements. This check contains:
 - Data elements exchanged by both ports shall have the same category (Value, Record, Array...)
 - Primitive data elements exchanged by both ports shall have the same name
 - In case of Record:
 - For each receiver sub element, there shall be a sender sub element with the same name:

E.g.:

 - > Sender of {element1; element2; element3} and receiver of {element1; element3} are compatible.
 - > Sender of {element1; element2} and receiver of {element1; element3} are **not** compatible.
 - > Sender of {element1} and receiver of {element1; element2} are **not** compatible.

- In case of Array, array data elements exchanged by both ports shall have the same name
- No conversion between data types is realized by the Rte in the generated code.



Note

Compatibility checks are about to be extended in DaVinci Developer and DaVinci Configurator Pro in the future releases.

5.2 Measurement and calibration

For specific ECU project purposes data elements may have to be accessed via measurement and calibration tools (MCD tools).

In order to generate an A2L file used by an MCD tool, the A2L generation has to be activated in the Rte configuration. This configuration is done in DaVinci Configurator Pro as shown Figure 5-2.

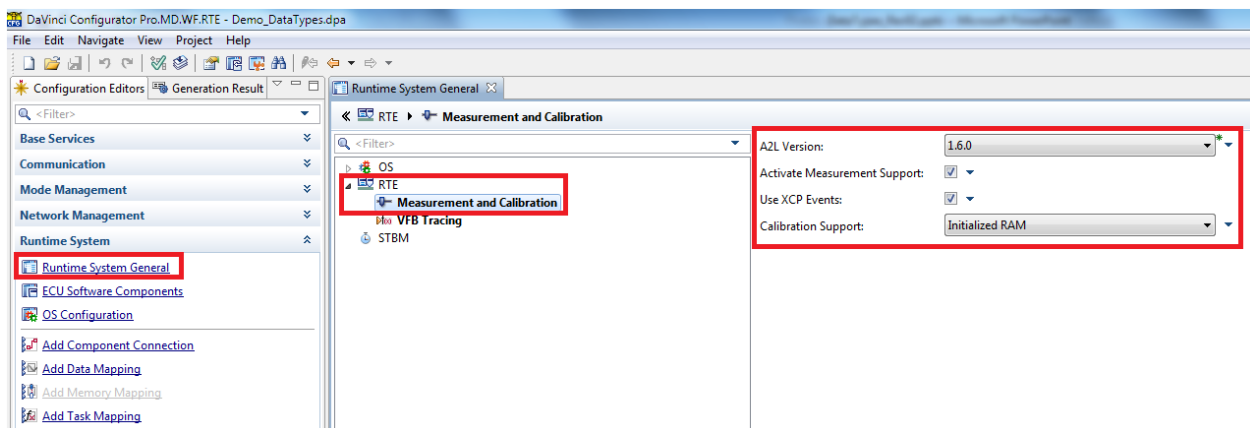


Figure 5-2 Rte configuration for A2L generation

A2L files are generated in /Config/McData/ folder of the ECU project (see Figure 5-3).

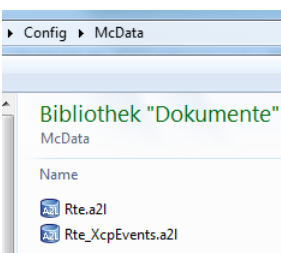


Figure 5-3 A2L generation folder

The following lines summarize what is generated and thus accessible from an MCD tool using the generated A2L file.

- ▶ The mode port prototypes are always visible in the A2L file. A text table is joined to it representing the conversion between internal value and mode declaration name.
- ▶ Data elements of Sender/Receiver interfaces are:
 - ▶ Not accessible if “NotAccessible” is set for the data element in the port interface editor.
 - ▶ Read only if “ReadOnly” is set for the data element in the port interface editor.
 - ▶ Read/Write if “ReadWrite” is set data element in the port interface editor.

**Note**

Measurement and calibration settings in application and implementation data types are over ridden by the setting at the data element in DaVinci Developer.

- ▶ Unit is accessible
- ▶ Conversion function and text tables are accessible:
 - ▶ 1:1 is set in case factor is 1 and offset 0 in DaVinci developer
 - ▶ Phys to Int factor Fp offset Op makes ($Fp \neq 0$): “int to phys $f(x) = (1/Fp) \cdot x - Op$ ” in A2L file
 - ▶ Int to Phys factor Fi offset Oi makes “int to phys $f(x) = Fi \cdot x + Op$ ” in A2L file

**Note**

Factors and offsets set for units in DaVinci Developer are not taken into account in A2L files.

- ▶ Operation arguments are not visible in the A2L even if explicitly set in DaVinci Developer.
- ▶ Calibration parameters:
 - ▶ They are accessible as set in DaVinci Developer (NotAccessible, ReadOnly or ReadWrite).
 - ▶ The unit is not available: the calibration parameter is only handled at implementation level with internal values.

5.3 Symbols

In DaVinci Developer certain elements may have a “Symbol” field. This is the case of implementation data types. A symbol field may be used in case the name of the element in the code shall be different from the name specified in DaVinci Developer.

For instance, the implementation data type created in chapter 4.5.1 can receive a Symbol name and value as done Figure 5-4. In this example, the name of the symbol is “MyImpl_Type_Ref_Symbol” and the value is “MyImpl_Type_Ref_Value”.

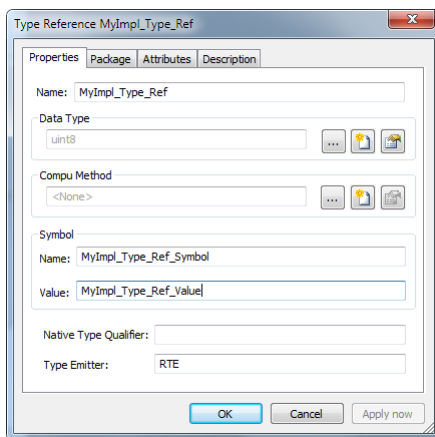


Figure 5-4 Symbol field for implementation data type

The corresponding code generated is:

In Rte_Type.h:

```
# define Rte_TypeDef_MyImpl_Type_Ref_Value
typedef uint8
```

The Value of the Symbol field is used as implementation data type name in the generated code.



Note

Symbols have a name field and a value field.

The name field is the name of the symbol element handled by the tools and the generators. The name that really appears in the code is what the value field contains.

5.4 Data type mapping assistant

A data type mapping assistant is available in DaVinci Developer. It is accessible by right click on the ECU Project root and by selecting “Data Type Mapping...” in the context menu (see Figure 5-5).

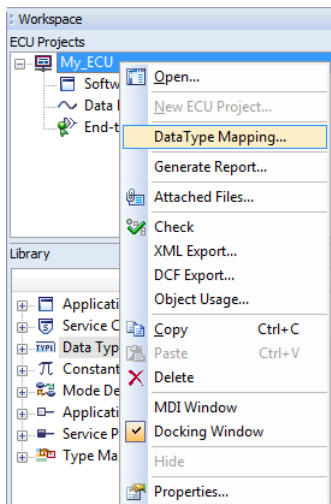


Figure 5-5 Data type mapping assistant opening

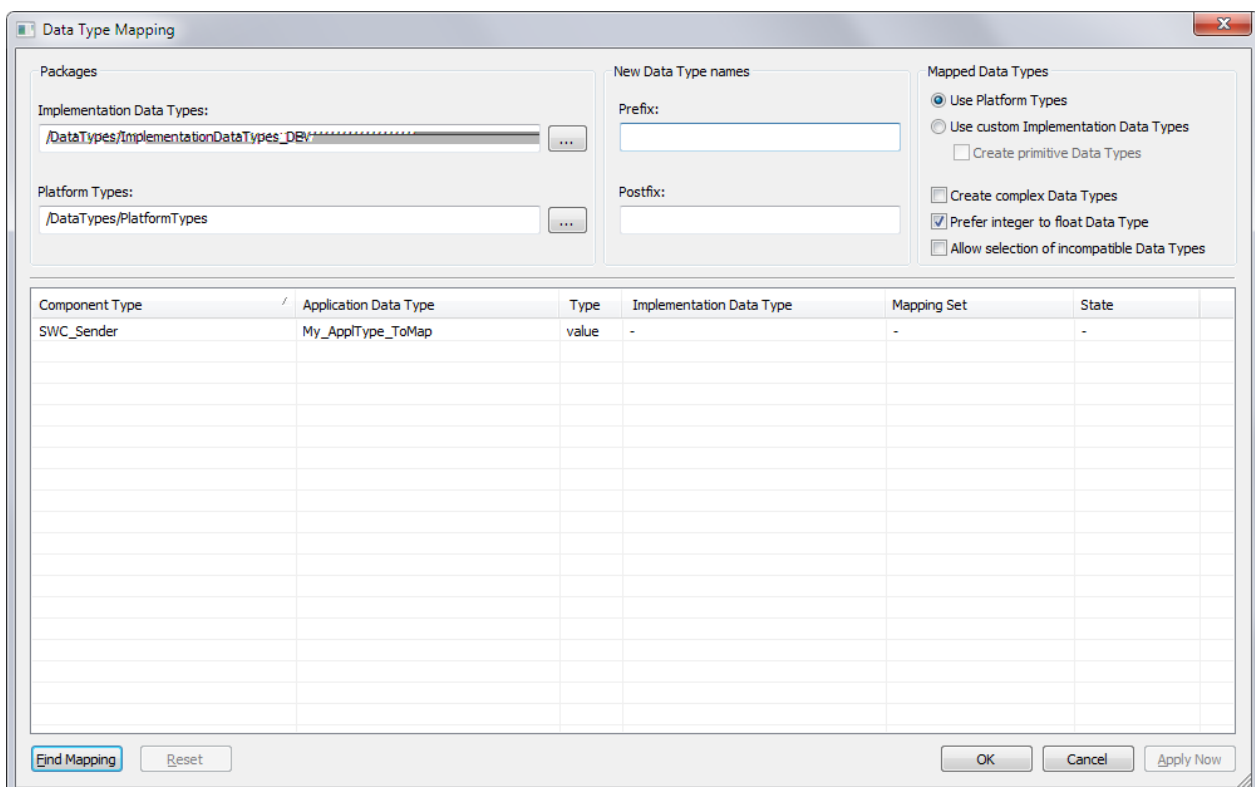


Figure 5-6 Data type mapping assistant

This assistant is a comfort editor allowing the mapping of application data types to implementation data types. In this editor it is possible:

- ▶ To let the assistant finding and mapping automatically compatible implementation data types using the “Find Mapping” button
- ▶ To let the assistant creating automatically a compatible implementation data types and mapping it using the “Find Mapping” button
- ▶ To create manually the mapping by right-clicking on the application data type line, choose “Map...” and select the implementation data type among the existing ones.

Note

The assistant is working only in the context of the ECU project. That means only application data types that are used in an ECU project will be displayed. If an application data type is only created in the library but not used, it will not appear in the list.

- > “Create complex Data Types”: lets the assistant creating new complex implementation data type
- > “Prefer integer to float Data Type”: lets the assistant mapping by preference to integer types than to float types.
- > “Allow selection of incompatible Data Types”: allows that non Autosar compatible data types are mapped
- > “Find Mapping”: starts to search for a compatible implementation data type
- > “Reset”: resets to the state when the editor was open

► 2 examples are presented here:

> 1st Example:

After pressing „Find Mapping“, the mapping assistant proposes to map with uint8 (see Figure 5-7). The state is set as “new” because it is a new proposal from the assistant. Pressing “ok” or “apply now” creates the mapping and changes the state to “exists”. The new mapping creation is reflected by Figure 5-9: a new data type mapping set with the default name “DataTypeMappings_DEV” is created where the application data type is mapped to uint8 and the corresponding SWC is referencing that data type mapping set.

Component Type	Application Data Type	Type	Implementation Data Type	Mapping Set	State
My_SWC	My_AppType_ToMap	value	uint8	-	new

Figure 5-7 New mapping

Component Type	Application Data Type	Type	Implementation Data Type	Mapping Set	State
My_SWC	My_AppType_ToMap	value	uint8	DataTypeMappings_DEV	exists

Figure 5-8 Existing mapping

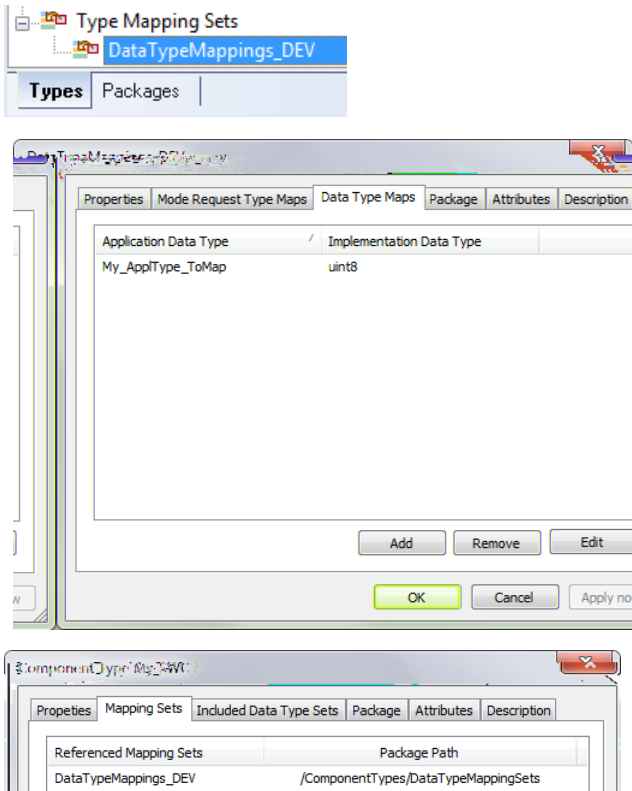


Figure 5-9 Created after new mapping

> 2nd example:

„Create primitive Data Types“ is checked. By clicking “Find Mapping” the state changes to “create data type” (see Figure 5-10).

Clicking “ok” or “apply now” a new implementation data type is created and mapped to the application data type. Its name starts with the “Prefix” field, then the application data type name and then the “Postfix” field. The state changes to “exists” (see Figure 5-11).

The new implementation data type can be found in the implementation data type library, the mapping in `DataTypeMappings_DEV`, which is referenced by the SWC (see Figure 5-12).

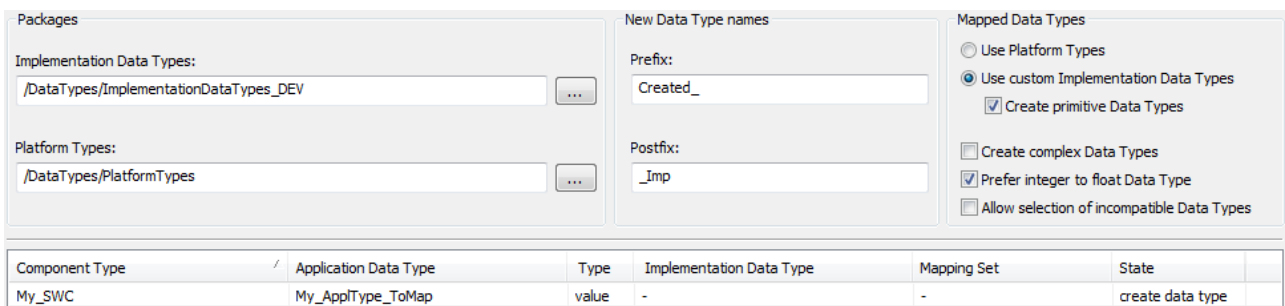
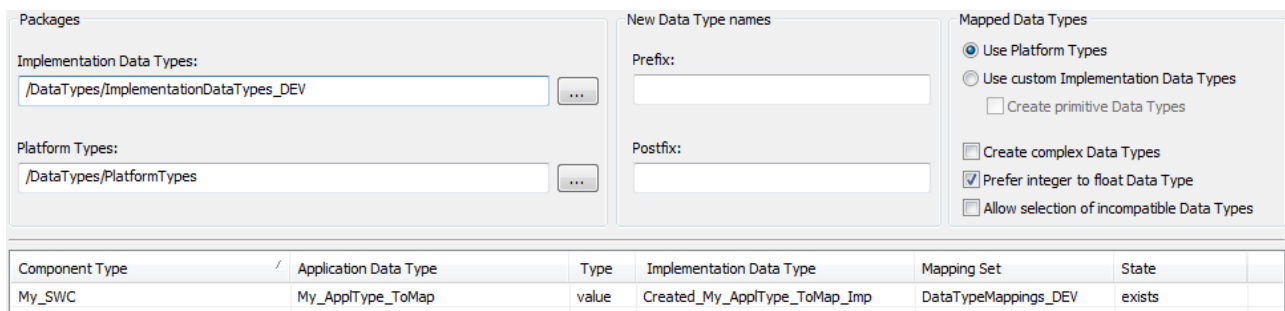
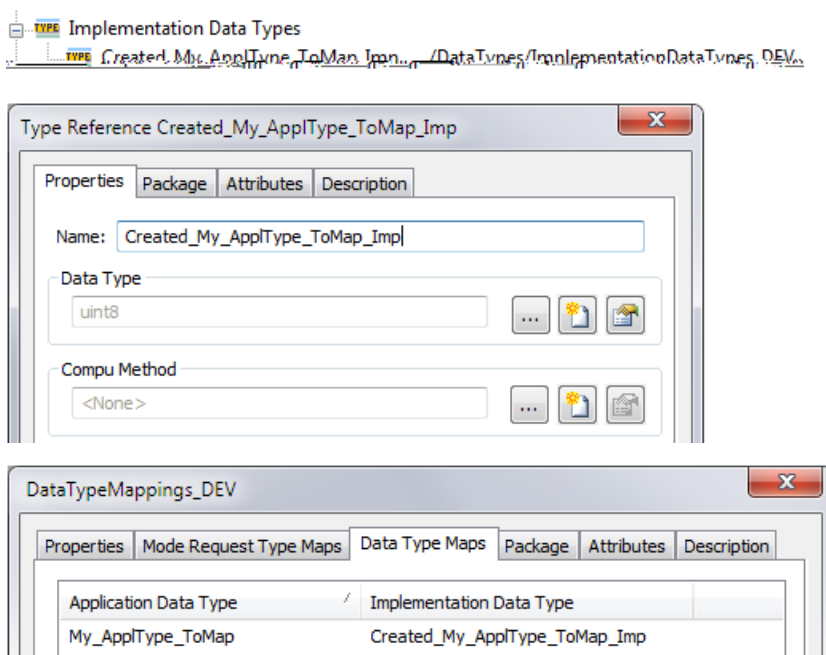


Figure 5-10 New mapping with implementation data type creation



Component Type	Application Data Type	Type	Implementation Data Type	Mapping Set	State
My_SWC	My_ApplType_ToMap	value	Created_My_ApplType_ToMap_Imp	DataTypeMappings_DEV	exists

Figure 5-11 Existing mapping with implementation data type creation



Application Data Type	Implementation Data Type
My_ApplType_ToMap	Created_My_ApplType_ToMap_Imp

Figure 5-12 Created after new mapping with implementation data type creation



Note

The assistant needs information from the application data type constraints and/or compu methods to be able to map it. In case this information is not sufficient, then the assistant will not be able to map automatically and the state will be “no mapping possible”:

State
no mapping possible

5.5 Type emitter

As defined by Autosar implementation data types have a “Type emitter” parameter. It defines the entity that shall provide the code definition of the implementation data type.

If Type emitter is set to “RTE” or is empty, the RTE shall generate a typedef for the implementation data type according to the native declaration of the referenced base type (category Value) or according to the referenced implementation data type (category Type Reference).

If Type emitter has any other value, then the RTE does not generate the corresponding typedef. It is here intended that the definition of that implementation data type will be provided by another entity.

6 Glossary and Abbreviations

6.1 Glossary

Term	Description

6.2 Abbreviations

Abbreviation	Description
SWC	Software Component
BswM	Basic Software Manager
ComM	Communication Manager
C	Programming language C

7 Contact

Visit our website for more information on

- > News
- > Products
- > Demo software
- > Support
- > Training data
- > Addresses

www.vector.com