Informatik	Professorship Computer Engineering **Automotive Software Engineering** Prof. Dr. Dr. h. c. Wolfram Hardt Dr. Batbayar Battseren	TECHNISCHE UNIVERSITÄT CHEMNITZ
Practical Unit 3	Designing an AUTOSAR application	Summer Semester 2025

Contents

1. Introduction

In this unit, you will the basics of AUTOSAR system design. This includes the definition of Software Components, Ports, Interfaces as well as Runnables which provide the actual functionality of a software component in the shape of a C function. The goal of this introduction is to apply the AUTOSAR knowledge from the "Automotive Software Engineering" lecture to build a simple AUTOSAR application. This application will then be simulated in software.
The following software programs from the automotive industry will be used:

• dSpace SystemDesk 5.6	for the System Design of the AUTOSAR application
• dSpace VEOS	for simulating the AUTOSAR application on a virtual ECU

2. AUTOSAR Basics

Since all AUTOSAR-related tools use the same terminology as defined in the AUTOSAR specification, let us quickly recapitulate the basics which should be known from the lecture.

2.1. Software Components

On the logical level an AUTOSAR application is made of software components (SWC). These SWCs are the most important elements to build AUTOSAR software. SWCs can contain complete functionalities or sub-functionalities of a system and communicate with other SWCs using ports. AUTOSAR defines different kinds of components, which can fulfil different kinds of tasks. A short overview of the most common software components is given in the Table 1.

Table 1: Most common types of SWCs as specified by AUTOSAR

Type	**Description**
Application Software Component	This kind of component contains the functionality or logic of an application. That's why it is the most important kind of component.
Sensor Software Component	Represents sensors and can communicate with the ECU Abstraction layer to retrieve sensor readings in order to provide them on its PPorts.
Actuator Software Component	Represents actuators and can communicate with the ECU Abstraction layer to control an actuators based on the data it receives over its RPorts.
ECU Abstraction Component	These can directly access the IO functions of an ECU. Consequently, they are able to access the Basic Software.
Complex Device Driver Component	CDD Components represent Complex Device Drivers, which can contain an application, but can also directly access the hardware.
Composition Component	Compositions contain one or more components connected through their ports.

Table 2: Interface types defined by AUTOSAR

Interface type	Description
Client-Server	A server provides functions (operations), which can be called by one or several clients.
Sender-Receiver	This kind of interface defines sender or receiver. The sender can send data (data elements) to one or several receivers.

2.2. Ports

Ports are needed for SWCs to communicate with other components.
There are mainly two different types of ports:

- Provided Ports (PPorts) which provide data as **output**
- Required Ports (RPorts) which require data as **input**

2.3. Interfaces

Interfaces define which kind of data (i.e. data types) a port should be able to receive or transmit and which communication methodology should be used. This means an interface is a binding contract between SWCs. Every port has one and only one interface assigned to it. But it is possible to assign one interface to several ports. Only ports which use the same interface can be connected to each other. There are different kinds of interfaces which are pre-defined by AUTOSAR, the two most commonly used ones are shown in Table 2. As the name implies the Client-Server interface is a straight implementation of the Client-Server communication methodology, while the Sender-Receiver interface is an implementation of the Producer-Consumer communication methodology. For this practical, we will only use the latter as it is the easiest to use.

2.4. Virtual Functional Bus (VFB)

In AUTOSAR, applications are built out of software compositions. And these software compositions are made of software components which are connected to each other through their defined ports. If two ports are connected the Virtual Functional Bus (VFB) takes care that data from the provided (output) port of one SWC flows to the required (input) port of the other SWC. At the end, the VFB is realized by the RTE (Runtime Environment) in software. The RTE provides necessary interfaces to communicate with other components. From the developers point of view, the actual communication technology used for this transmission (e.g. CAN bus, Ethernet, WiFi) does not matter - especially in early development phases.

2.5. Summary

With regards to AUTOSAR in this practical, you should keep the following things in mind:

- Software Components (SWCs) are the central structure of AUTOSAR applications
- Every SWC needs ports in order to communicate
- There are different kinds of SWCs for different purposes
- Interfaces describe a binding contract between ports/SWCs
- Every port has one and only one interface

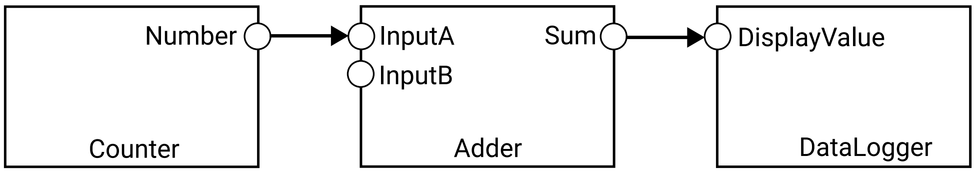
3. Practical Tasks

SystemDesk is a very complex development tool which abstracts away most of the low-level configuration and setup which is specified in the AUTOSAR standard. In the end, the build output of SystemDesk is the source code of an AUTOSAR application including all required configuration files (arxml). This means as part of the build step the RTE is generated and BSW modules are configured and prepared. This application can then be compiled to build an actual application which can be flashed or simulated.

In this tutorial, we will create a simple software architecture whose task is to take 2 numbers add them and print the result. We will realize this task using 3 SWCs:

- Counter: will continuously generate numbers through incrementing
- Adder: will take two numbers and sum them
- DataLogger: will display any number supplied to it

The corresponding final target architecture is illustrated below:



Please note that we will intentionally leave the second input of the Adder unconnected. The relevance of this will be elaborated on later.

For the practical, we will follow the **AUTOSAR methodology**. This means we will define the design (i.e. components, interfaces & structure) of the AUTOSAR project **first**. Only when this is finished, we will implement the actual functionality of the SWCs. In addition, we will follow a step-wise approach to allow early testing of the functionality. As such, we will first build a solution with only the Counter and DataLogger SWC. And if this is working we will add the Adder SWC.

Throughout this tutorial, important information will be highlighted in boxes like this one. Make sure to follow all steps carefully especially with regard to the **naming of components**. As AUTOSAR will generate lots of source code for you, this will only work if the names of your components are correct / match.

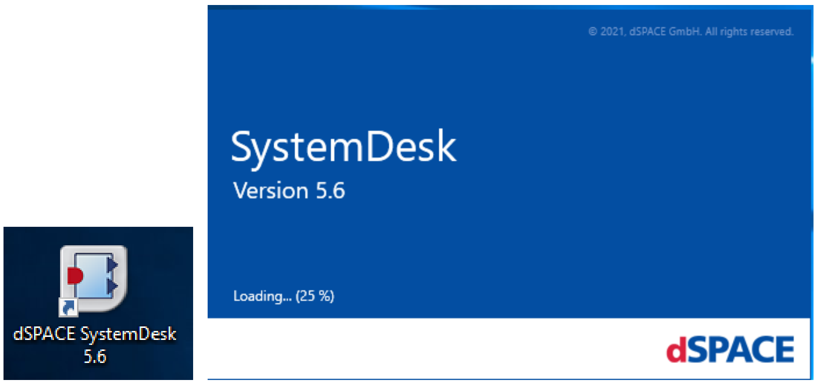
3.1. Practical Task 1: Creating an AUTOSAR Project

In task 1 you will:

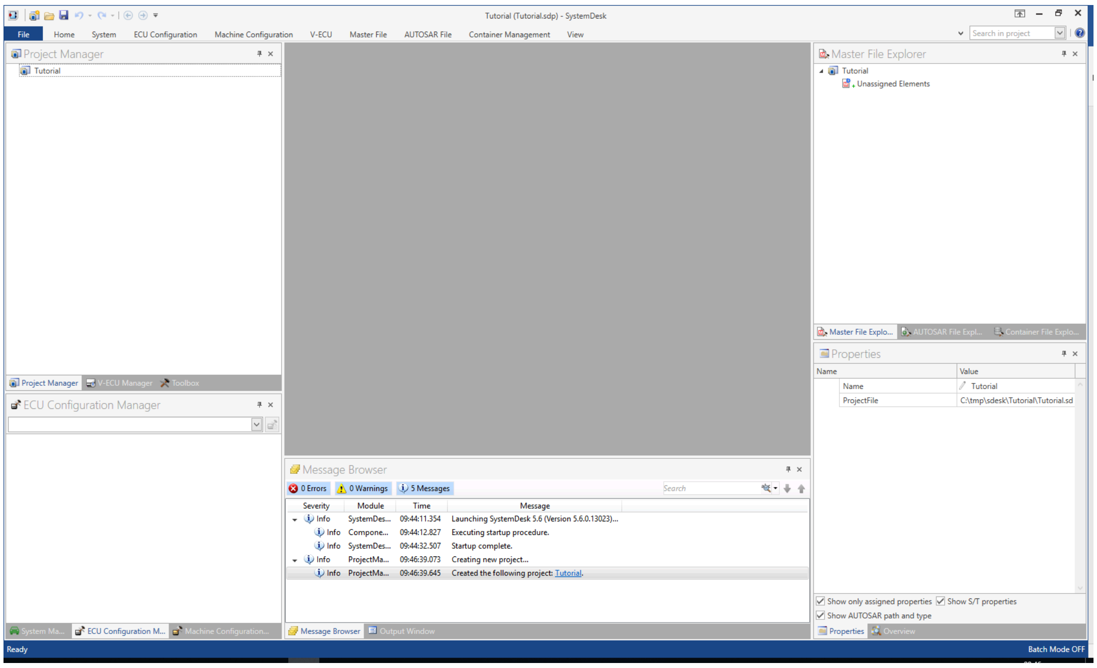
- Create a new project in dSpace SystemDesk
- Import AUTOSAR standard datatypes

3.1.1. Starting SystemDesk

On the practical PCs there is shortcut to start SystemDesk on the desktop. Use it to start SystemDesk which may take a while (if it has not been started yet).



After SystemDesk started you will see its main window:



Message Browser

0 Errors | 0 Warnings | 5 Messages

```
Severity   Module        Time           Message
Info       SystemDes...  09:44:11.354   Launching SystemDesk 5.6 (Version 5.6.0.13023)...
  Info     Compone...    09:44:12.827   Executing startup procedure.
  Info     SystemDes...  09:44:32.507   Startup complete.
Info       ProjectMa...  09:46:39.073   Creating new project...
  Info     ProjectMa...  09:46:39.645   Created the following project: Tutorial.
```

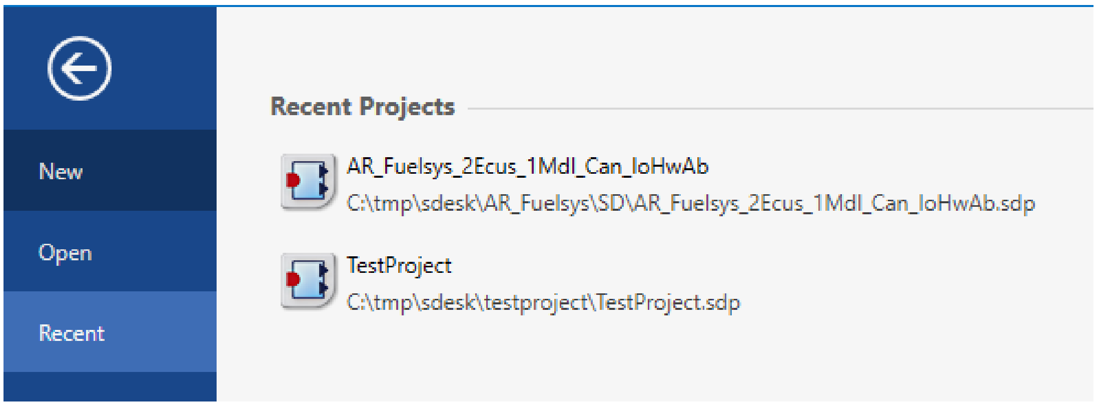
Properties

Name	Value
Name	Tutorial
ProjectFile	C:\tmp\sdesk\Tutorial\Tutorial.sd

3.1.2. Creating a New Project

Click on *File* -> *New* to create a new project. Name the project "Tutorial".

Save the new project in a folder called "Unit3" on the Desktop of the Lab PC. Usually, you should always save SystemDesk projects in an empty folder as SystemDesk (and you) will create more files and folders alongside the project file later.

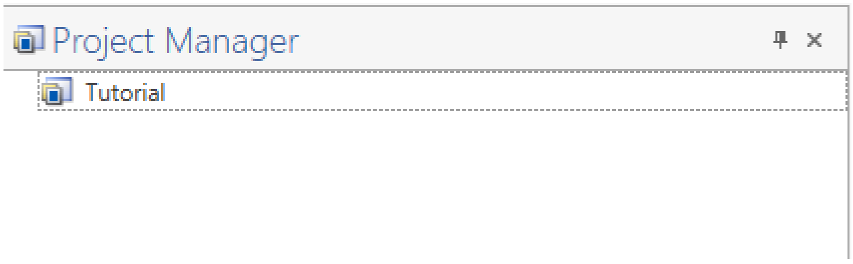


Recent Projects

AR_Fuelsys_2Ecus_1Mdl_Can_IoHwAb
C:\tmp\sdesk\AR_Fuelsys\SD\AR_Fuelsys_2Ecus_1Mdl_Can_IoHwAb.sdp

TestProject
C:\tmp\sdesk\testproject\TestProject.sdp

When the project has been created successfully, the Project Manager tool window will look like this:

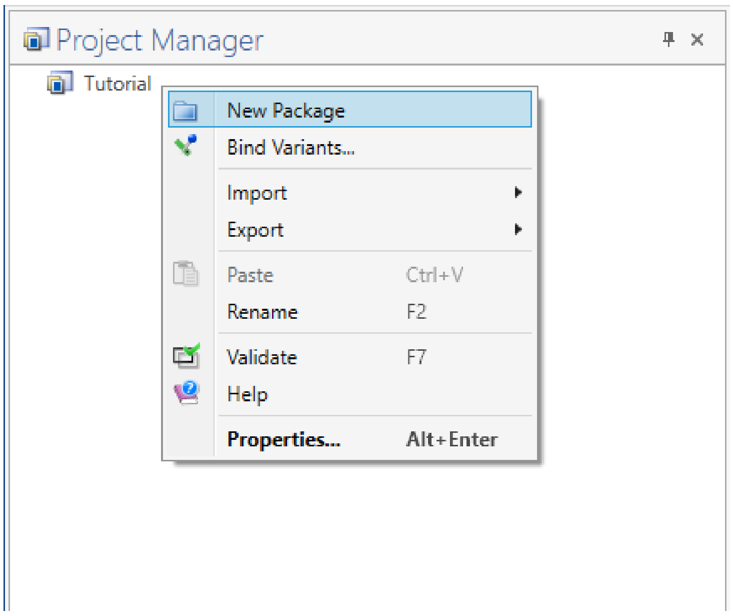


We can now start to design the AUTOSAR application.

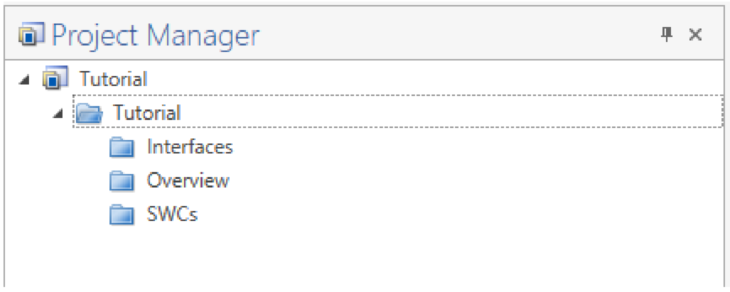
3.1.3. Preparing the Project Structure

Right-click on Tutorial and then select New Package to create a Project package (Displayed as a new folder). Name this package “Tutorial” then create three sub-folders named “Interfaces”, “SWCs” and “Overview”.
“Interfaces” is going to contain all interfaces which are needed. “SWCs” is going to contain all the Software Components and “Overview” all the composition diagrams.

While it is not strictly necessary to separate the project into packages, it makes the project clear and sorted which is a must for very large projects as used in the automotive industry.

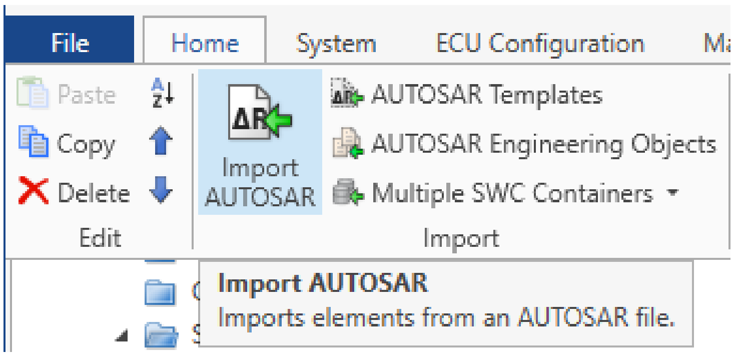


After this step the library should look like this:



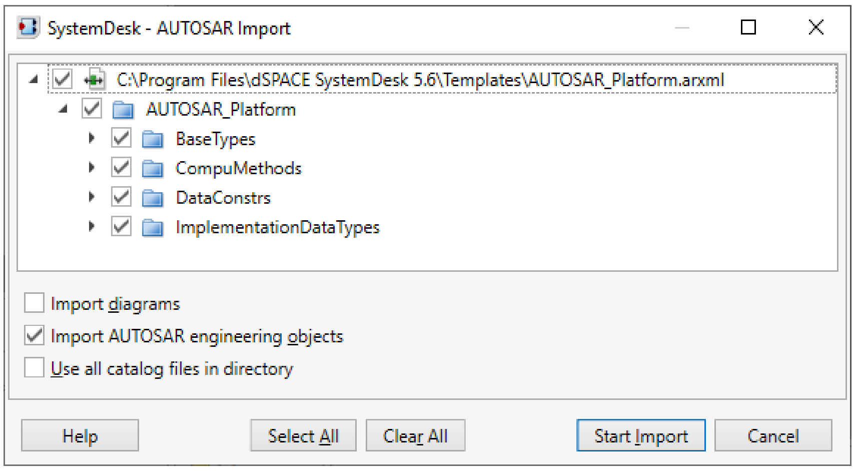
3.1.4. Importing Data Types

Since we want to develop software in an hardware-independent way we cannot use the normal C datatypes (e.g. int, float, …) in our implementation as the actual size of these depends on the architecture of the target ECU. Fortunately, AUTOSAR already defines a set of base data types with platform-independent sizes and behavior. An application can use these data types as base types. As a new project contains no datatypes at all, we need to import these primitive data types through an AUTOSAR standard ".arxml" file. This can be done by clicking on *Import AUTOSAR* in the file menu.



Please choose the following .arxml file in the file dialog:
"C:\Program Files\dSPACE SystemDesk 5.6\Templates\AUTOSAR_Platform.arxml"

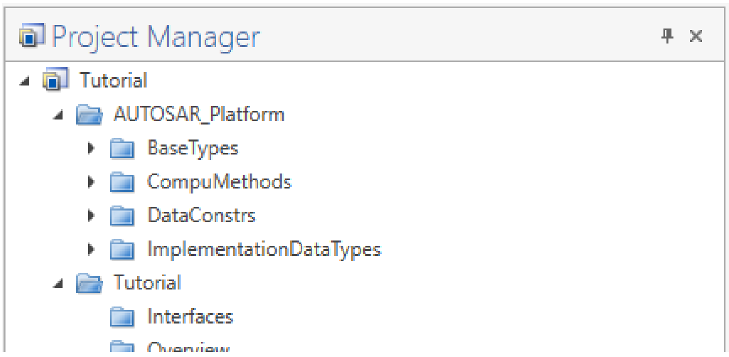
Afterwards the "AUTOSAR Import" window will be shown. Make sure that all elements are selected and that "Import AUTOSAR engineering objects" is ticked.


SystemDesk - AUTOSAR Import

- ✓ C:\Program Files\dSPACE SystemDesk 5.6\Templates\AUTOSAR_Platform.arxml
  - ✓ AUTOSAR_Platform
    - ✓ BaseTypes
    - ✓ CompuMethods
    - ✓ DataConstrs
    - ✓ ImplementationDataTypes

- [ ] Import diagrams
- [✓] Import AUTOSAR engineering objects
- [ ] Use all catalog files in directory

By clicking on "Start Import" a new package is created and standard data types are made available in the project. For this practical, this will allow us to use the *uint8* datatype in our design and source code, which represents a single byte (unsigned 8-bit integer, value from 0 to 255).



You have now finished setting up a new project and are ready to add actual components to it.

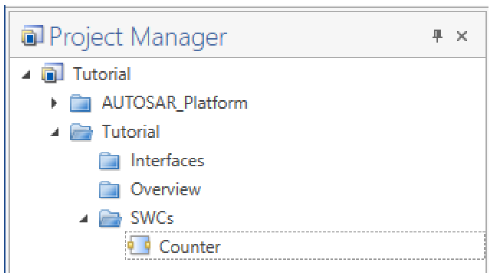
3.2. Practical Task 2: Defining a New Software Component

Based on the project you created in the previous task, in this task you will learn how to:

- Create a new Software Component
- Add a port to it
- Assign interfaces to a SWC's ports
- Define the functionality container (Internal Behavior) and Runnabele of the SWC
- Setup an event which triggers a Runnable

3.2.1. Creating a Software Component (SWC)

Now we can create the first software component "Counter". Right-click on the SWCs folder and select: *New -> Application SW Component Type*. A new component will be added to the library.

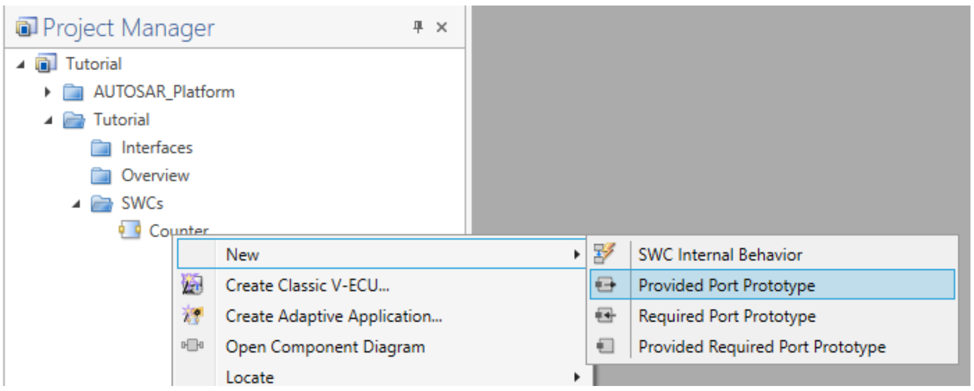


Every component should have a unique, desrcriptive name. In this example, you should name the component “Counter”, because it is going to generate numbers by counting upwards.

In general, the name of a component is usually pre-defined during system design. As the name will influence the name of the generated code functions and architecture files it is very important **to keep it consistent**. General names like System, Status or standalone reserved words from the AUTOSAR standard (Port, RTE etc.) should be avoided.

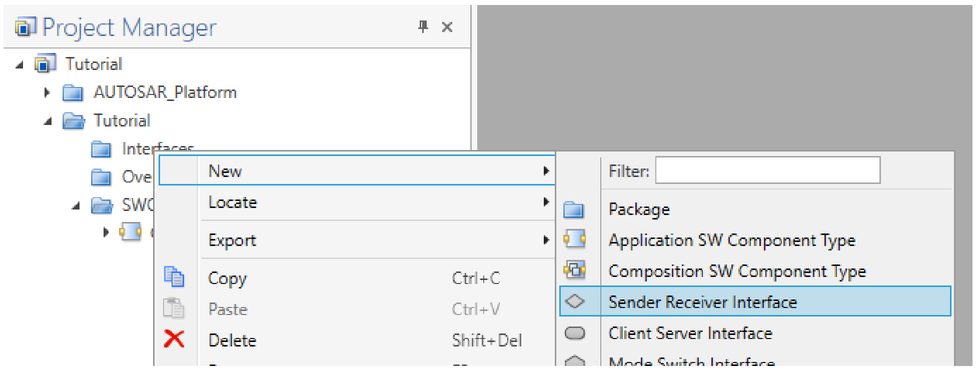
3.2.2. Creating a Port

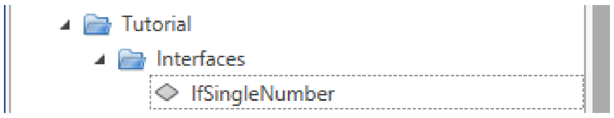
The main task of the Counter component is to supply the AUTOSAR program with input numbers. Consequently, it needs to transmit the generated numbers to the connected SWCs. As mentioned in Section 1, a port is necessary to communicate. To create an output port right-click the component and select *New -> Provided Port Prototype*. A new port should appear. Please name this port “Number”.



3.2.3. Creating an Interface

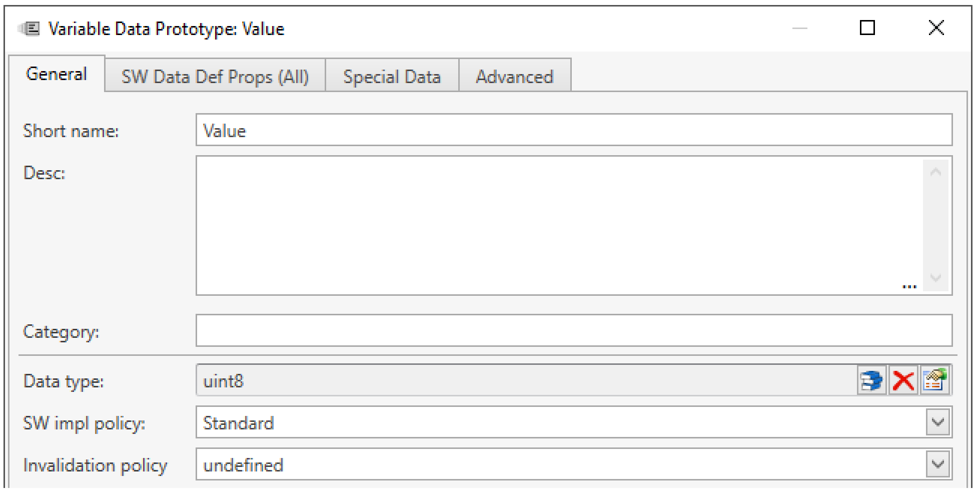
The next step is to assign an interface to the port. As mentioned in Section 2.3 an interface is a contract which describes the kind of data that can be received or transmitted. For this example, the port should receive an 8-bit unsigned integer, i.e. a **single byte**.
When creating an interface, the first step is to think about the type of communication needed for the application task. Since we do not want to request data but need a continuous flow of the current value, a Sender-Receiver interface is used. To create a Sender-Receiver interface, right-click on the “Interfaces” folder and select *New -> Sender/Receiver Interface*. A new interface will appear. Often the abbreviation “If” is used to identify interfaces, so name it “IfSingleNumber”.





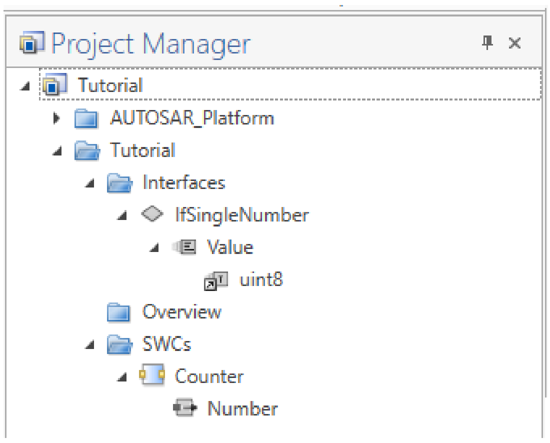
An interface defines a container of data which is to be exchanged between ports. To define the actual data elements contained in this interface, right-click on the interface and select *New -> Data Element*. The name of the data element should be "Value". Double click the newly created element to open its properties. Please select **uint8** as data type (unsigned 8-bit integer).

It is possible to define as many data elements per interface as needed. In C terms, you can think of the interface as a structure definition. With regard to this tutorial, only one data element is used.



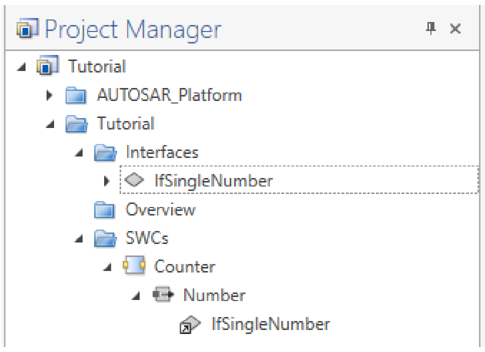
Short name:	Value
Desc:	
Category:	
Data type:	uint8
SW impl policy:	Standard
Invalidation policy	undefined

Afterwards the new elements should be present in the Project Manager.



3.2.4. Assigning Interfaces

Now the interface must be assigned to the port. To assign an interface to a port, simply drag the interface onto the desired port. In this case "IfSingleNumber" has to be dragged on "Number". To visualize this, the interface will be added as a child element to the port.



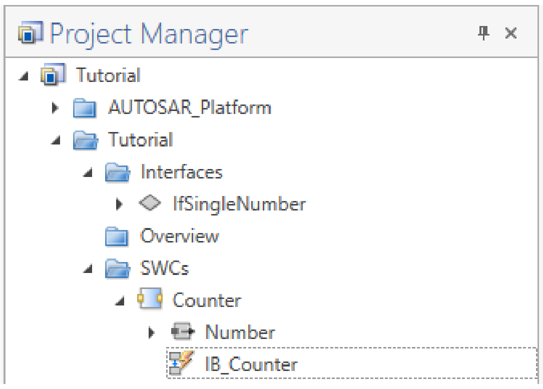
Two ports can only exchange data with each other when they both have been assigned the same interface.

3.2.5. Adding Internal Behavior

The actual functionality which is realized by an SWC is called Internal Behavior in the AUTOSAR terminology. The Internal Behavior acts as a container for application logic (realized through source code, models, etc.).

To add an Internal Behavior right-click on the software component in the Project Manager and select *New* -> *SWC Internal Behavior*. The internal behavior name should be unique. It is recommended to use the name of the component prefixed with "IB_".

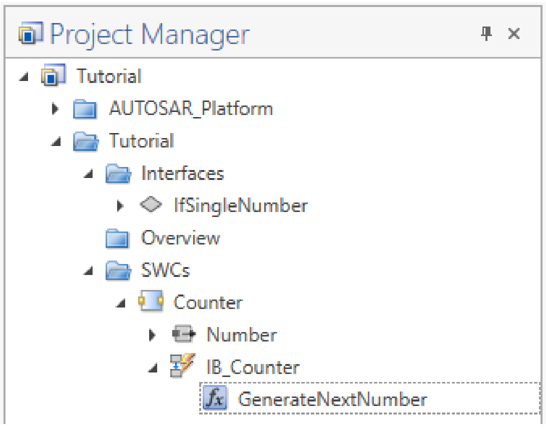
Consequently, in our case it is named "IB_Counter".



3.2.6. Adding Runnables

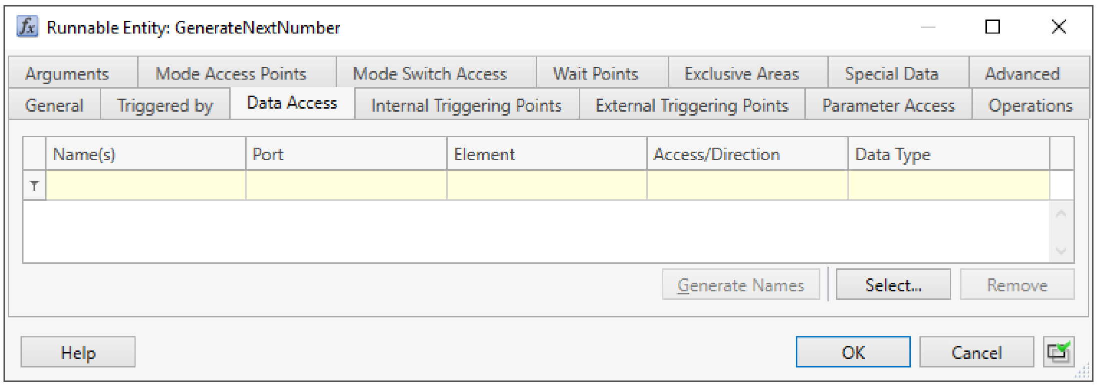
The next step is to add a Runnable. Runnables are functions, which contain the actual application logic (= functionality) of the software components and will be contained by Internal Behavior. Often, a Runnable is a C function which is called by AUTOSAR based on a pre-defined event.
For the Counter, we want to add a Runnable which is executed every 500ms (i.e. twice a second) to generate new data.

To add a Runnable, right-click on the Internal Behavior and select, *New -> Runnable Entity*. Use "GenerateNextNumber" as name.

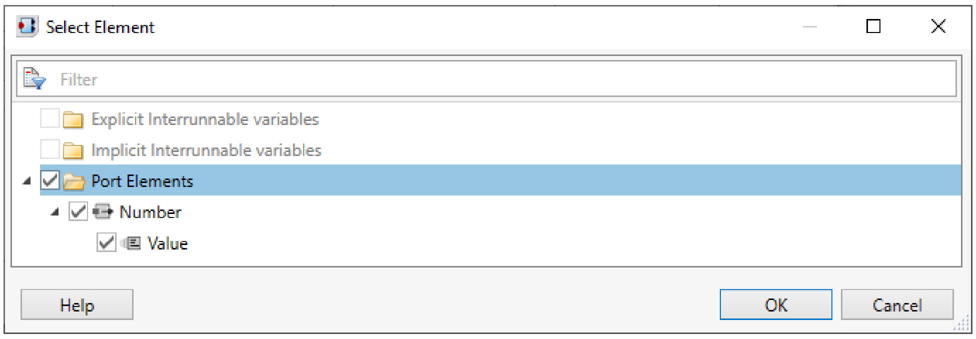


Subsequently, double-click on the Runnable and its properties window will be opened. Select the "Data Access" tab. In this tab it can be defined which ports (and corresponding data elements of the assigned interfaces) can be accessed by the Runnable. Hence, click on the "Select" button and check the box of the data element, which is to be accessed by this Runnable. In this case, the data element "Value" should be checked. Now the runnable can access this data element.

A Runnable can only access data of ports if this has been **explicitly allowed**. If Data Access has not been set, no port accessor functions will be generated by AUTOSAR. Consequently, if you try to access an unconfigured port in your Runnable this will lead to compile-time errors.


Runnable Entity: GenerateNextNumber

Name(s)	Port	Element	Access/Direction	Data Type

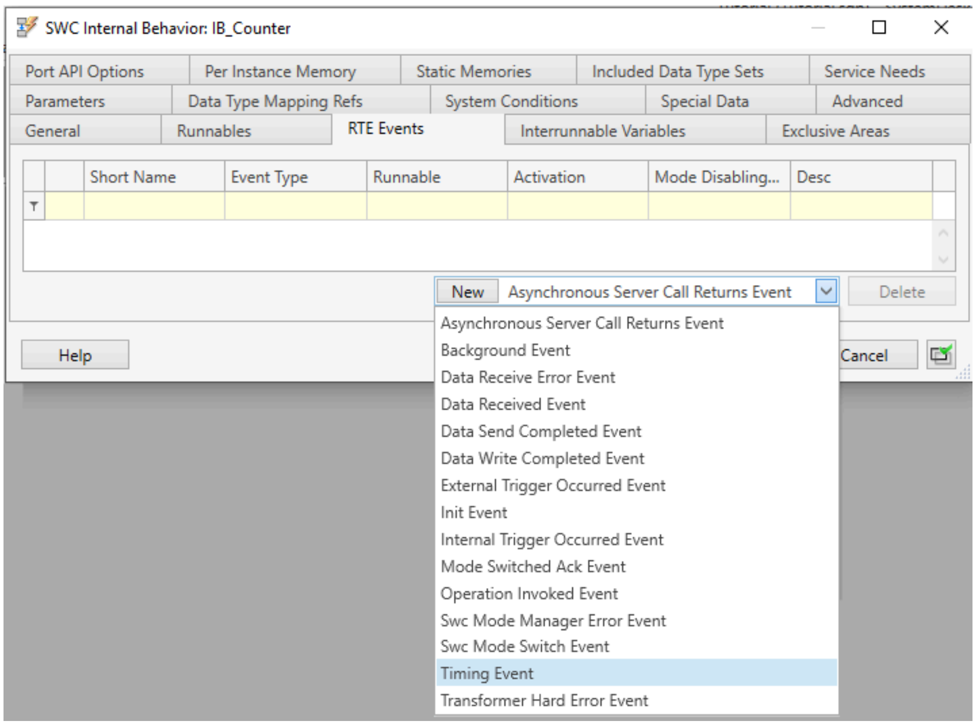

Select Element

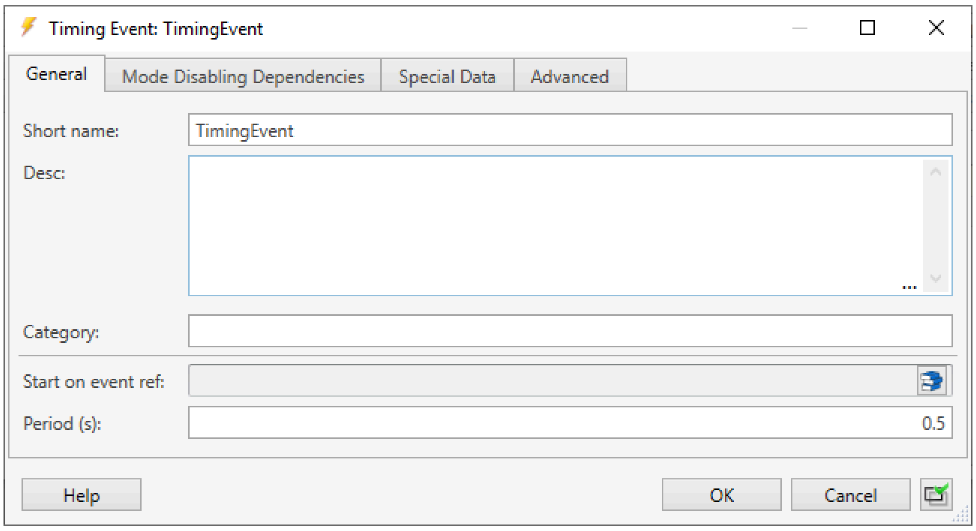
Filter
Explicit Interrunnable variables
Implicit Interrunnable variables
Port Elements
Number
Value
Help
OK
Cancel

3.2.7. Creating an RTE event

In order for a Runnable to be executed during the runtime of the AUTOSAR application, it **must** be triggered by an RTE event.

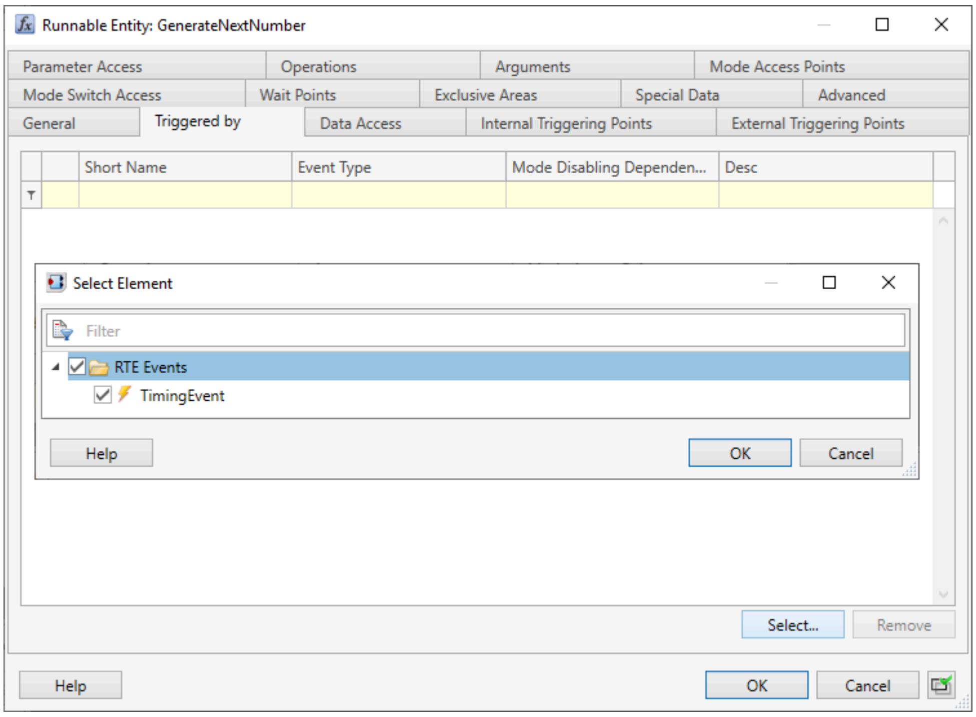
To set this up, double-click on the internal behavior ("IB_Counter"). The properties window of the Internal Behavior is opened. Switch to the "RTE-Events" tab. Here, events can be defined, which trigger Runnables at runtime. In our case, we want the Runnable to be run every 500ms (i.e. twice a second). To achieve this, select "Timing Event" at the bottom of the window and click on "New". Double click the newly created event in the table to open its properties and input „0.5" (= 500 ms) for Period as you have to provide the value in seconds.

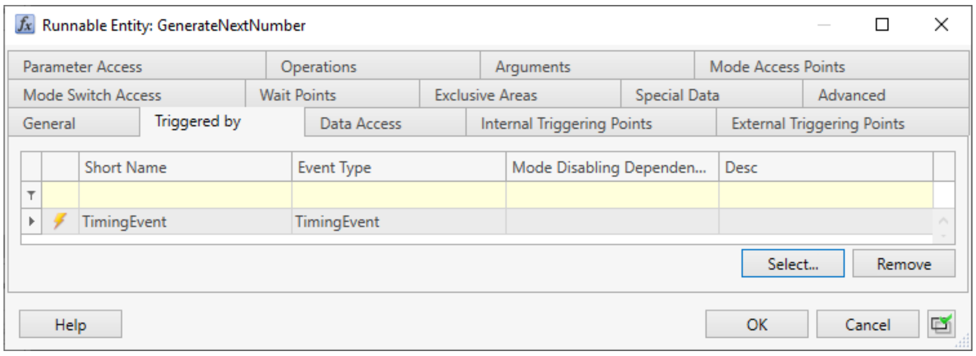




Short name:	TimingEvent
Desc:	
Category:	
Start on event ref:	
Period (s):	0.5

After creating an event it needs to be assigned to a Runnable which it should trigger. To do this, close the properties window of the internal behavior and open the properties windows of the Runnable again. Go to the “Triggered by” tab and there click “Select…”, check the created event and confirm with OK. The event assignment should now be visible in the table.



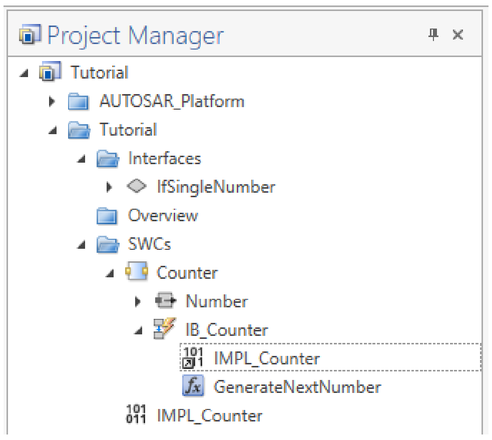


Short Name	Event Type	Mode Disabling Dependen...	Desc
TimingEvent	TimingEvent		

A Runnable will only be executed when it is triggered by an RTE event. If you have Runnables in your project which have not been assigned at least one RTE event this will generate a **validation error** and you **cannot** build the AUTOSAR project.

3.2.8. Adding an Implementation Placeholder

An Internal Behavior contains Runnables and their corresponding Implementation. Implementation describes the actual logic which should be executed when the corresponding Runnable is triggered by an event. Among others this can be C source code or Simulink models. In the practical, we will use C functions. To add an Implementation, right-click on the Internal Behavior and select *New -> SWC Implementation*. The name of the implementation should be unique again. Here the prefix "IMPL_" is recommended, so name it "IMPL_Counter".
As we are still in the design phase, we will not write or attach any source code now. Consequently, it is not necessary to configure anything in the implementation at this point.


Project Manager
Tutorial
AUTOSAR_Platform
Tutorial
Interfaces
IfSingleNumber
Overview
SWCs
Counter
Number
IB_Counter
IMPL_Counter
GenerateNextNumber
IMPL_Counter

3.3. Practical Task 3: Completing the Application Design

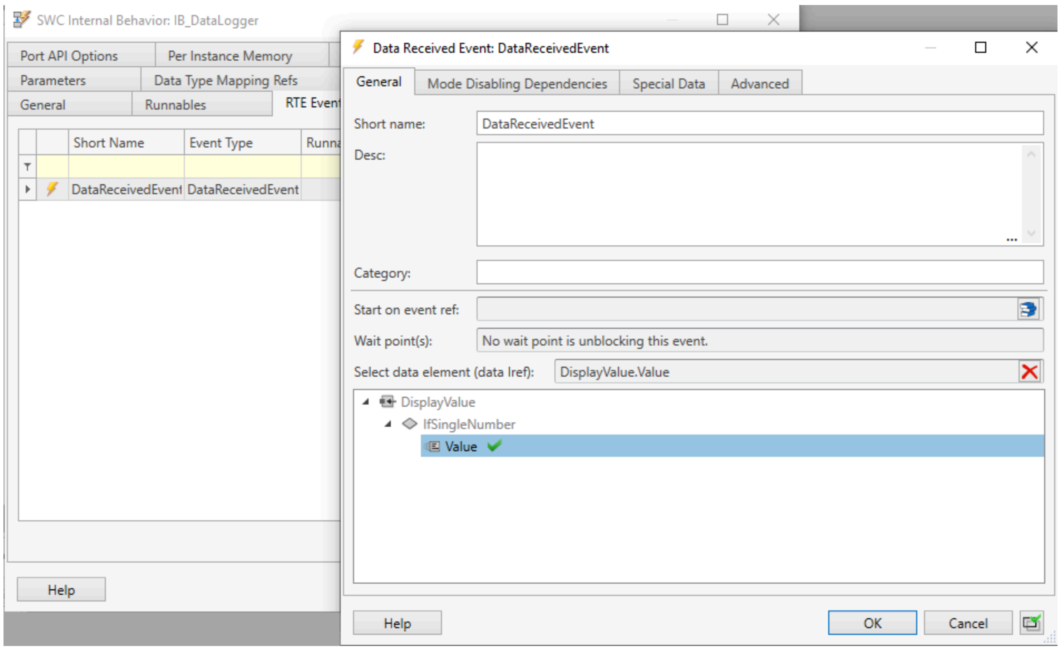
Based on the knowledge you gained in task 1, in task 2 you will:

- Create a second SWC including port definitions and internal behavior
- Setup a corresponding event to trigger execution
- Compose both SWCs you created to an actual application by connecting their ports
- Create template source code files for both SWCs

3.3.1. Adding a Second SWC

As you now know how SWCs are added to an AUTOSAR System Architecture, how ports are setup and Internal Behavior is defined, **please conduct the following steps to finish the intermediate architecture**:

1. Add a second SWC named "DataLogger"
   - Add a Required Port to it named "DisplayValue"
   - Assign the interface IfSingleNumber to the new port
   - Add Internal Behavior
   - Add a Runnable called "Log"
   - Add an SWC Implementation
2. Configure "Data Access" of the Runnable "Log" in such a way, that the DisplayValue port can access the corresponding Value data element
3. Add an RTE Event for the DataLogger's Runnable
   - Add a "Data Received Event" in the "RTE Events" tab of IB_ProcessingComponent
   - Select the Value element from the DisplayValue port (see Figure)



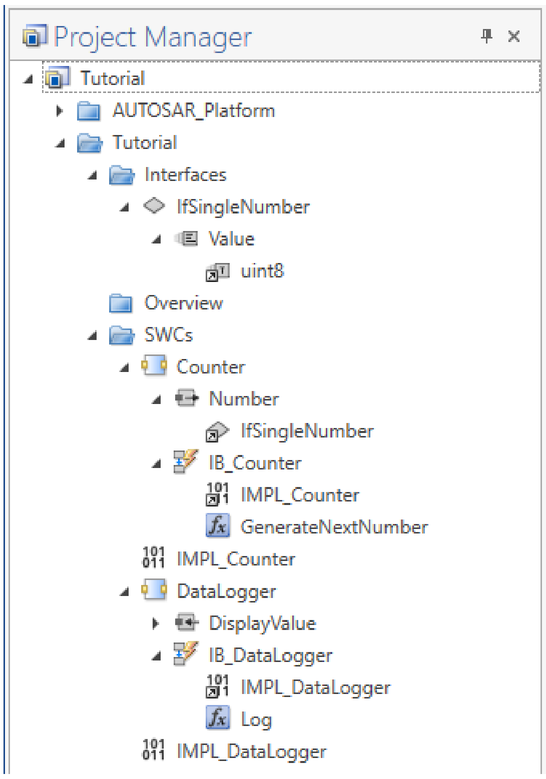
Short Name	Event Type	Runna
DataReceivedEvent	DataReceivedEvent	

Short name: DataReceivedEvent
Desc:
Category:
Start on event ref:
Wait point(s): No wait point is unblocking this event.
Select data element (data Iref): DisplayValue.Value

   - Set this Event as trigger for the Log Runnable
     - This means, every time data is received on this port we want AUTOSAR to execute the Log Runnable

Remember:
It is vital to configure "Data Access" of Runnables after adding new ports. Otherwise, the Runnables **cannot read or write** data from the ports which will later lead to **compile-time errors as required functions are not generated.**

After adding the required elements, the project should look as follows:

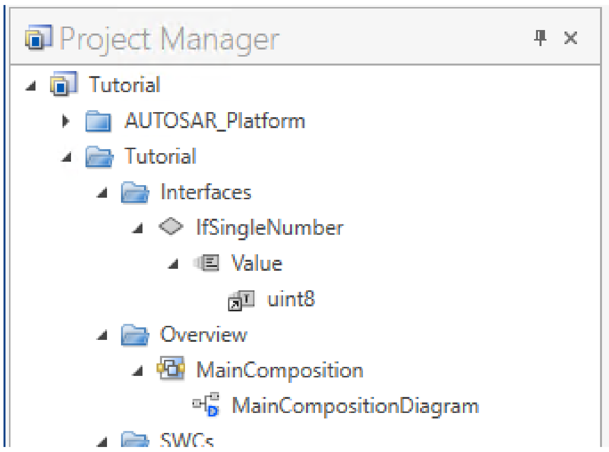


```
Project Manager
Tutorial
    AUTOSAR_Platform
    Tutorial
        Interfaces
            IfSingleNumber
                Value
                    uint8
        Overview
        SWCs
            Counter
                Number
                    IfSingleNumber
                IB_Counter
                    IMPL_Counter
                    GenerateNextNumber
            IMPL_Counter
            DataLogger
                DisplayValue
                IB_DataLogger
                    IMPL_DataLogger
                    Log
            IMPL_DataLogger
```

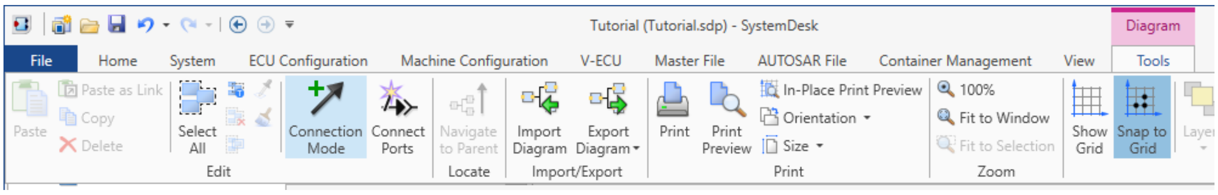
3.3.2. Composing the Application

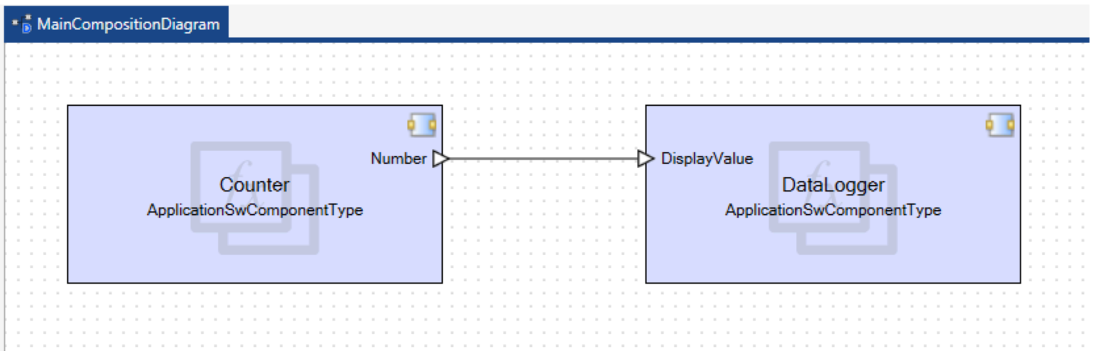
Since you have finished defining the individual components, we can now design the actual application from these. We followed a Bottom-Up-approach in this tutorial. Often, the system architecture is designed first with placeholder SWCs and ports / interfaces. Then the implementation of the individual components starts against these contracts.

To define our application, a Composition Software Component must be created which contains a description of the logical architecture of our application. Right click on the "Overview" Package and create a new "Composition SW Component Type" and rename it to "MainComposition". This component must contain a composition diagram which can be created by right clicking on "MainComposition" and selecting *New* -> *Empty Composition Diagram* from the context menu. This new diagram should be named "MainCompositionDiagram". You can think of this diagram as the entry point for your AUTOSAR application.



Now, simply double click on MainCompositionDiagram which open it presenting an empty white area in the middle of the window. Drag and drop your Application SWCs from the Project Manager to this space. Each SWC will be represented as a functional block with inputs (required ports) and/or outputs (provided ports). Now connect the Number port of the Counter SWC to the DisplayValue port of the DataLogger SWC. Do this by going to the Diagram Tools section in the window's main ribbon band and clicking on "Connection Mode". Then click & drag a line between the ports. Only ports with a matching interface can be connected.



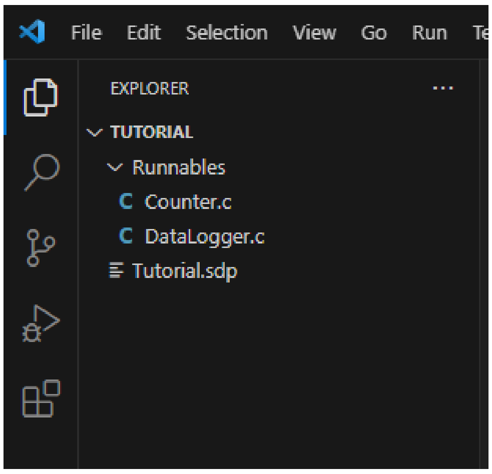


You have now finished the design of the application. However, at this point we cannot build the AUTOSAR project yet as we have not supplied any implementation (in the form of C source code). We will do so in the next (final) step by using empty, template source code files.

3.3.3. Providing Template Source Code Files

After completing the architecture, source code files must be assigned to the respective implementations of the SWCs. In our case, the behavior of each SWC is defined in a single C file. As we are still in the System Design phase, we will only create stub implementation which do nothing. The actual implementation will be done at a later point. As our project contains 2 Runnables we need to provide 2 corresponding C functions. As **SystemDesk is not an IDE** we need to do this in an external code editor. In this practical we will use **Visual Studio Code** as it provides a good integration for code completion later on.
Consequently, start Visual Studio Code and open the folder your project is situated in as

workspace. We want to structure our project and keep the runnable implementations together in one place. For this Right-click in Visual Studio Code's explorer and create a new folder called "Runnables".



In this Runnable folder create two files called "Counter.c" and "DataLogger.c" which will house the C functions for the Runnable implementations. Based on AUTOSAR's naming conventions, we can now create two empty stub implementations as follows:

Content of Counter.c :

```
#include <Rte_Counter.h>

void Counter_GenerateNextNumber()
{

}
```

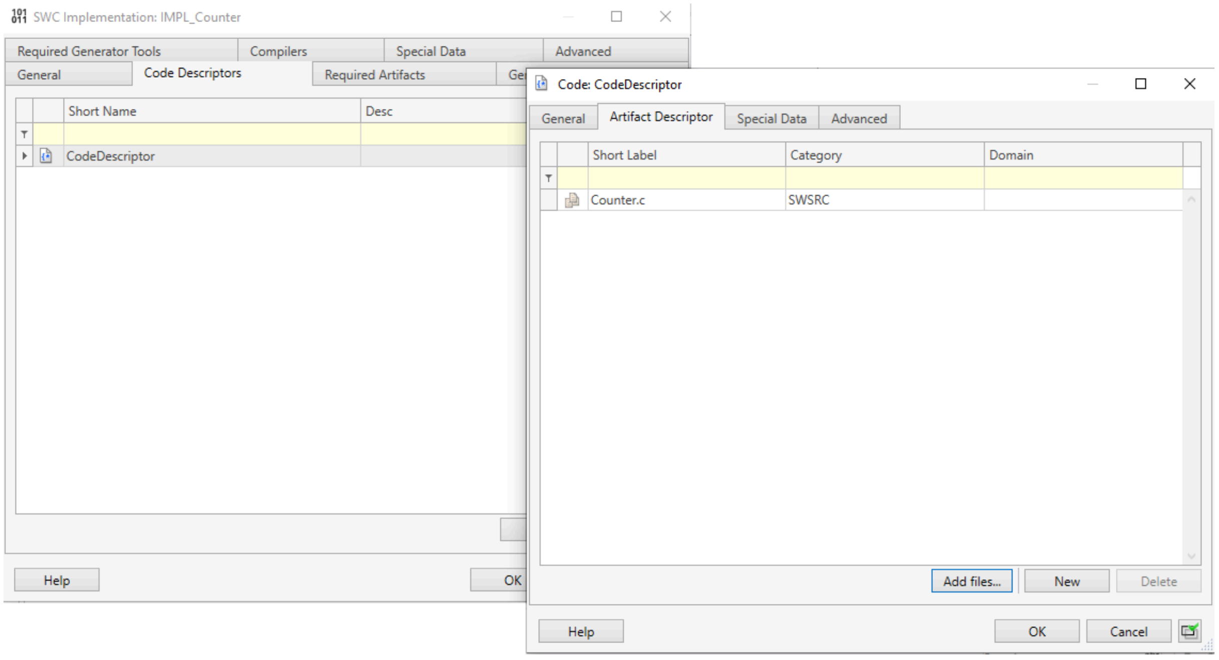
Content of DataLogger.c :

```
#include <Rte_DataLogger.h>
#include <Sab.h>

void DataLogger_Log()
{

}
```

Since the RTE has not been created yet, Visual Studio Code will probably complain about that the included header files are missing (i.e. red squiggly lines). You can ignore this for now as we will generate the RTE in the next section.

As we now have our template code files, we have to assign each Runnable the corresponding source code file: To do so go to the implementation of the corresponding software component. Here it is the "Counter" software component. Double click on *IMPL_Counter* to open its properties. Go to the *Code Descriptors* Tab and click on *New*. Double click on the newly created row in the table named "CodeDescriptor". Now a new Window opens. There you have to select the *Artifact Descriptor* Tab. Click on "Add files.." and select the Counter.c file in your file system.

Then confirm the dialogs with *OK*. Add the DataLogger.c to the DataLogger in the same fashion. Now everytime, when a Runnable is triggered AUTOSAR will execute the corresponding C function.



Short Name	Desc
CodeDescriptor	

Short Label	Category	Domain
Counter.c	SWSRC	

Congratulations, you finished creating your Application Design. Now we can start with the configuration of the target platform (i.e. target ECU) in order to generate an RTE which will provide us with methods to read and write data from and to ports.

3.4. Practical Task 4: Generating the RTE

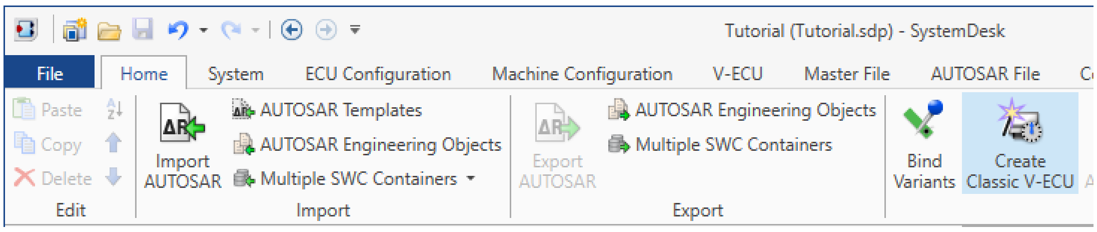
Now that we have completed our (hardware-independent) system design. We need to map the application to the actual target platform, i.e. one or multiple ECUs. For the practical, we will use a single ECU which contains the whole application. We will create a so-called Virtual ECU in order to demonstrate that AUTOSAR applications can be tested in software without the need for actual ECU hardware.

In this task, you will:

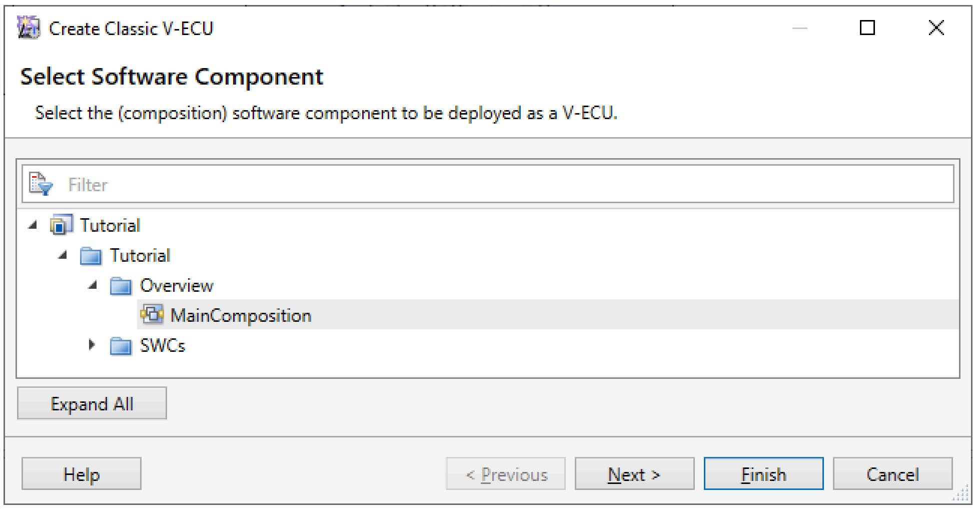
- Create a Virtual ECU for executing the AUTOSAR application without a hardware ECU
- As part of this, generate the RTE (and BSW) for your system

3.4.1. Creating a Virtual ECU

Fortunately, SystemDesk makes it very easy to create a (basic) virtual ECU. Switch to the Home tab in the main ribbon band and click on "Create Classic V-ECU" to create a virtual ECU for testing purposes.



In the new window you have to select the MainComposition SWC which should be deployed to the new virtual ECU then click Next. If your currently active view in SystemDesk is a Composition Diagram, often this is pre-selected and the corresponding step is skipped.


Create Classic V-ECU

Select Software Component

Select the (composition) software component to be deployed as a V-ECU.

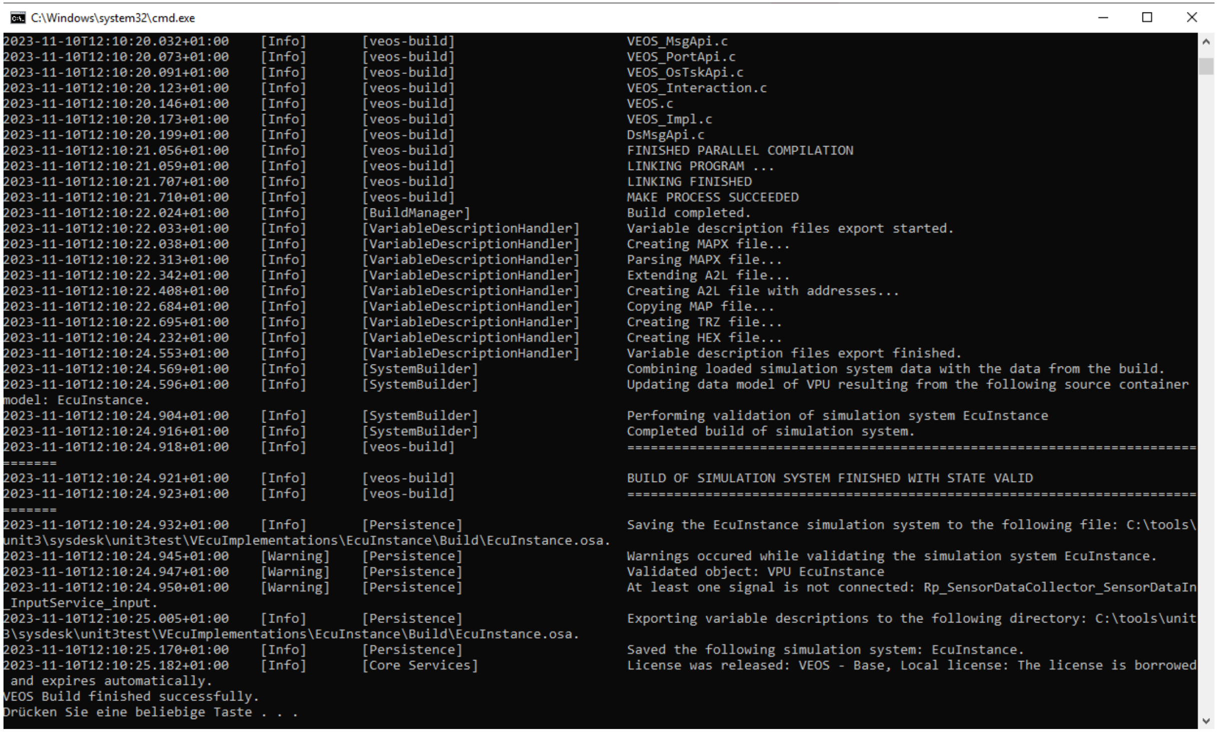
As we do not want to change any options you can directly click on Finish.

If SystemDesk does not show the *Select Software Component* step, make sure that no other SWC is selected in the project manager as this is chosen automatically for building the virtual ECU otherwise.

SystemDesk will now create the v-ECU by generating the code files for the AUTOSAR system, i.e. the RTE + BSW, which will take some time.

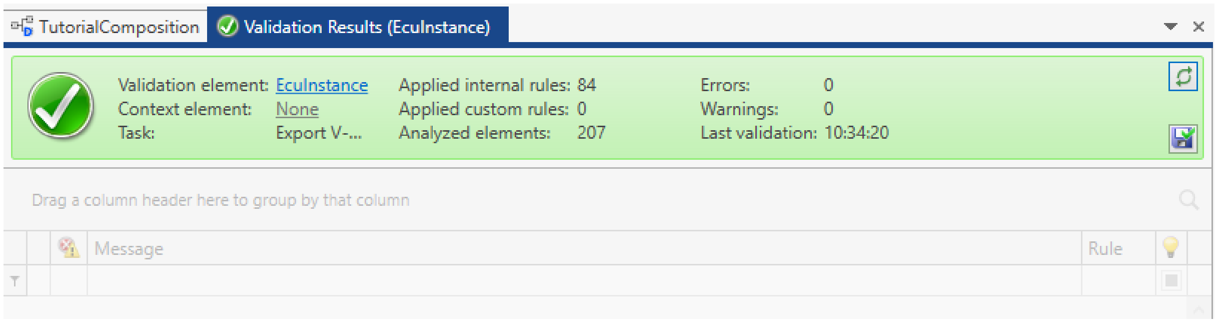
If the build process does not start automatically, please click the “Build” action in the V-ECU tab of the main window’s toolbar

During the build, a console window will open which shows the generation progress. Once it finishes, please check that it states “VEOS Build finished successfully”. If this is not the case, you most likely have a configuration error in your project (unassigned interfaces, ports, data type mismatches, …)



```
2023-11-10T12:10:20.032+01:00    [Info]       [veos-build]                     VEOS_MsgApi.c
2023-11-10T12:10:20.073+01:00    [Info]       [veos-build]                     VEOS_PortApi.c
2023-11-10T12:10:20.091+01:00    [Info]       [veos-build]                     VEOS_OsTskApi.c
2023-11-10T12:10:20.123+01:00    [Info]       [veos-build]                     VEOS_Interaction.c
2023-11-10T12:10:20.146+01:00    [Info]       [veos-build]                     VEOS.c
2023-11-10T12:10:20.173+01:00    [Info]       [veos-build]                     VEOS_Impl.c
2023-11-10T12:10:20.199+01:00    [Info]       [veos-build]                     DsMsgApi.c
2023-11-10T12:10:21.056+01:00    [Info]       [veos-build]                     FINISHED PARALLEL COMPILATION
2023-11-10T12:10:21.059+01:00    [Info]       [veos-build]                     LINKING PROGRAM ...
2023-11-10T12:10:21.707+01:00    [Info]       [veos-build]                     LINKING FINISHED
2023-11-10T12:10:21.710+01:00    [Info]       [veos-build]                     MAKE PROCESS SUCCEEDED
2023-11-10T12:10:22.024+01:00    [Info]       [BuildManager]                   Build completed.
2023-11-10T12:10:22.033+01:00    [Info]       [VariableDescriptionHandler]     Variable description files export started.
2023-11-10T12:10:22.038+01:00    [Info]       [VariableDescriptionHandler]     Creating MAPX file...
2023-11-10T12:10:22.313+01:00    [Info]       [VariableDescriptionHandler]     Parsing MAPX file...
2023-11-10T12:10:22.342+01:00    [Info]       [VariableDescriptionHandler]     Extending A2L file...
2023-11-10T12:10:22.408+01:00    [Info]       [VariableDescriptionHandler]     Creating A2L file with addresses...
2023-11-10T12:10:22.684+01:00    [Info]       [VariableDescriptionHandler]     Copying MAP file...
2023-11-10T12:10:22.695+01:00    [Info]       [VariableDescriptionHandler]     Creating TRZ file...
2023-11-10T12:10:24.232+01:00    [Info]       [VariableDescriptionHandler]     Creating HEX file...
2023-11-10T12:10:24.553+01:00    [Info]       [VariableDescriptionHandler]     Variable description files export finished.
2023-11-10T12:10:24.569+01:00    [Info]       [SystemBuilder]                  Combining loaded simulation system data with the data from the build.
2023-11-10T12:10:24.596+01:00    [Info]       [SystemBuilder]                  Updating data model of VPU resulting from the following source container
model: EcuInstance.
2023-11-10T12:10:24.904+01:00    [Info]       [SystemBuilder]                  Performing validation of simulation system EcuInstance
2023-11-10T12:10:24.916+01:00    [Info]       [SystemBuilder]                  Completed build of simulation system.
2023-11-10T12:10:24.918+01:00    [Info]       [veos-build]                     ========================================================================
=======
2023-11-10T12:10:24.921+01:00    [Info]       [veos-build]                     BUILD OF SIMULATION SYSTEM FINISHED WITH STATE VALID
2023-11-10T12:10:24.923+01:00    [Info]       [veos-build]                     ========================================================================
=======
2023-11-10T12:10:24.932+01:00    [Info]       [Persistence]                    Saving the EcuInstance simulation system to the following file: C:\tools\
unit3\sysdesk\unit3test\VEcuImplementations\EcuInstance\Build\EcuInstance.osa.
2023-11-10T12:10:24.945+01:00    [Warning]    [Persistence]                    Warnings occured while validating the simulation system EcuInstance.
2023-11-10T12:10:24.947+01:00    [Warning]    [Persistence]                    Validated object: VPU EcuInstance
2023-11-10T12:10:24.950+01:00    [Warning]    [Persistence]                    At least one signal is not connected: Rp_SensorDataCollector_SensorDataIn
_InputService_input.
2023-11-10T12:10:25.005+01:00    [Info]       [Persistence]                    Exporting variable descriptions to the following directory: C:\tools\unit
3\sysdesk\unit3test\VEcuImplementations\EcuInstance\Build\EcuInstance.osa.
2023-11-10T12:10:25.170+01:00    [Info]       [Persistence]                    Saved the following simulation system: EcuInstance.
2023-11-10T12:10:25.182+01:00    [Info]       [Core Services]                  License was released: VEOS - Base, Local license: The license is borrowed
 and expires automatically.
VEOS Build finished successfully.
Drücken Sie eine beliebige Taste . . .
```

After completing it will show the Validation Results of the created ECU, hopefully without errors.

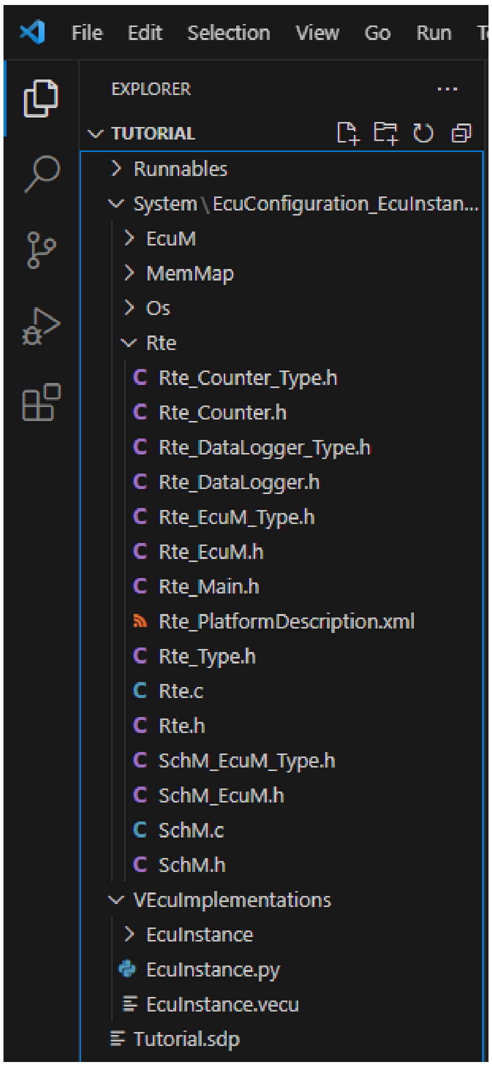

TutorialComposition
Validation Results (EcuInstance)
Drag a column header here to group by that column
Message
Rule

Validation element: EcuInstance
Context element: None
Task: Export V-...
Applied internal rules: 84
Applied custom rules: 0
Analyzed elements: 207
Errors: 0
Warnings: 0
Last validation: 10:34:20

If there are errors in your case, please **read them carefully**. Usually, SystemDesk states the cause for validation errors very clearly and provides quick links to the property windows of the components where the problem can be fixed. In this tutorial, an error most likely means that you forgot a step while creating the system design.

If you check the directory of your project (e.g. in Visual Studio Code) you will see that SystemDesk has created two folders:

- *System*, which contains the generated source code files of the RTE and AUTOSAR application
- *VEcuImplementations* which contains the created Virtual ECU



```
> Runnables
v System \ EcuConfiguration_EcuInstan...
  > EcuM
  > MemMap
  > Os
  v Rte
    Rte_Counter_Type.h
    Rte_Counter.h
    Rte_DataLogger_Type.h
    Rte_DataLogger.h
    Rte_EcuM_Type.h
    Rte_EcuM.h
    Rte_Main.h
    Rte_PlatformDescription.xml
    Rte_Type.h
    Rte.c
    Rte.h
    SchM_EcuM_Type.h
    SchM_EcuM.h
    SchM.c
    SchM.h
v VEcuImplementations
  > EcuInstance
  EcuInstance.py
  EcuInstance.vecu
Tutorial.sdp
```

For each Software Component a corresponding RTE header file will be generated. In our case "Rte_Counter.h" and "Rte_DataLogger.h". This header file can then be used in the implementation of the Runnable to read and write ports. (We already included the files in our code templates since we knew there were being generated.)

3.5. Practical Task 5: Implementing SWC Functionality

You have now successfully generated an RTE and AUTOSAR application for a virtual ECU. However, we left the Runnable's C functions as empty in order to be able to generate the RTE. Now is the time to complete them.
Switch back to Visual Studio Code. Since we now have an RTE which contains methods to access ports of our SWCs, code completion will be available in the source code files of the Runnables. This means, VS Code will provide you with help regarding the available RTE functions you can use in your components while you type.

Sometimes Visual Studio Code will still show errors regarding missing files. In this case just restart it, it will then discover the generated RTE automatically.

Remember, that the generated functions depend on your configuration. So if for instance, you did not provide a Runnable access to a specific port the corresponding read or write function for this port will be missing.

Now, please implement the following functionality for the SWCs:

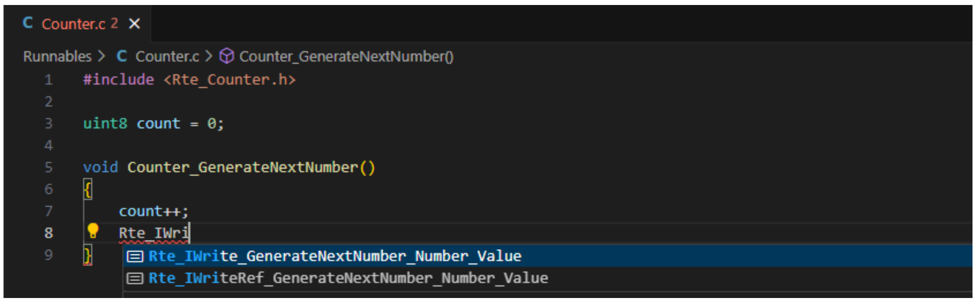
We have configured, that the Counter's Runnable is called periodically through a timed event. Every time the function is invoked it should increase an **internal, global counter variable** by 1 and output the new value on its out port. To write to a port please use the RTE function which is named based on AUTOSAR's naming conventions as shown in the lecture:

```
Rte_IWrite_[RunnableName]_[PortName]_[DataElementName]([value to write]);
```

So, as the Runnable is named **GenerateNextNumber**, the port is named **Number** and the data element of the interface is called **Value** the call for this would be:

```
Rte_IWrite_GenerateNextNumber_Number_Value( <<your counter variable>> );
```

Visual Studio Code will assist you with code completion for typing the long Rte call names:



```
#include <Rte_Counter.h>

uint8 count = 0;

void Counter_GenerateNextNumber()
{
    count++;
    Rte_IWri
}
```

```
Rte_IWrite_GenerateNextNumber_Number_Value
Rte_IWriteRef_GenerateNextNumber_Number_Value
```

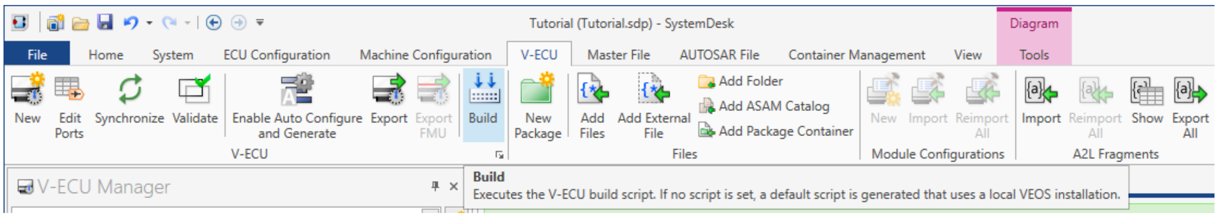
We have configured the DataLogger's Runnable to be run every time data on its RPort is received. In this tutorial, the component should just read the incoming data value and log it to the simulator's log window. Use the following RTE function to read from a port (make sure to use the correct datatype of the interface you are reading):

```
uint 8 value = Rte_IRead_[RunnableName]_[PortName]_[DataElementName]();
```

Use the function `Sab_SubmitInfo` (which is declared in the header file `Sab.h` we already included in DataLogger.c) to print the value you received, it works similar to the normal C printf function, e.g.:

```
Sab_SubmitInfo("Received data: %d", value);
```

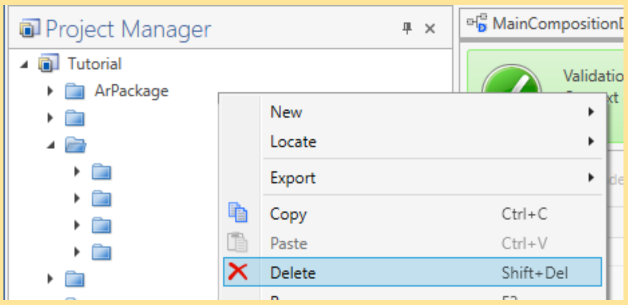
As our system design is already completed, you can test the compilation of your implementation by switching to SystemDesk and just clicking on “Build” in the V-ECU tab.



Build
Executes the V-ECU build script. If no script is set, a default script is generated that uses a local VEOS installation.

Compilation errors, if any, will be shown in the build output window. If there are no errors, you can proceed to the next section and test your implementation.

Every time you make changes to the design or configuration of your AUTOSAR application (i.e. adding or assigning ports, interfaces, components). You have to rebuild the Virtual ECU, as the RTE needs to be re-generated. To do this, is to **delete** the **ArPackage** which contains the created EcuInstance & System nodes in the “Project Manager”. Then repeat the “Create Classic V-ECU” flow.



In contrast to this, if you **only** changed source code it is enough to click *Build* in SystemDesk without re-creating the Virtual ECU.

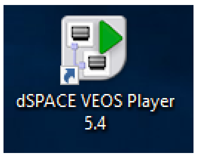
3.6. Practical Task 6: Running and Testing the AUTOSAR Application in a Simulator

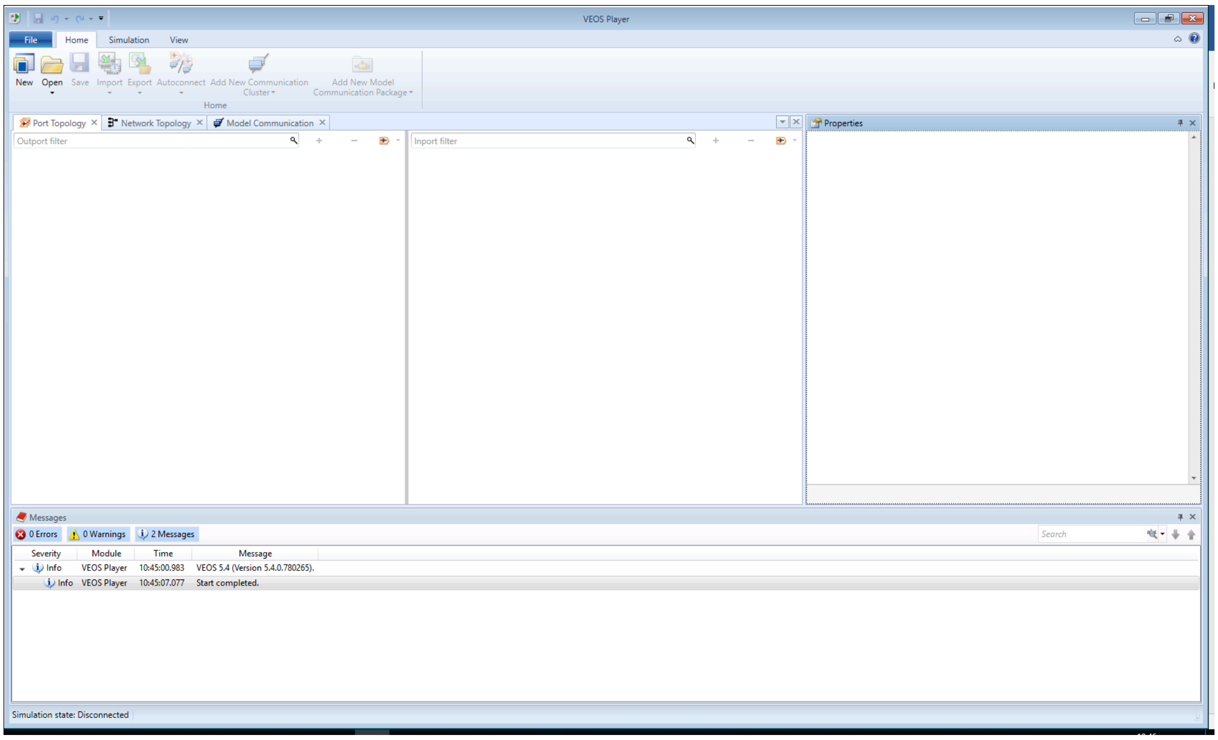
You have now finished designing and implementing a basic AUTOSAR application. Additionally, you have mapped it to a Virtual ECU for which RTE and BSW have been generated. In this task you will:

- Use dSpace VEOS Player to execute your implemented AUTOSAR application
- Check the simulation output to see if your application works as desired

3.6.1. Starting dSpace VEOS

Simulating AUTOSAR applications is handled by the software dSpace VEOS Player. You can start it with the corresponding desktop shortcut or through the Windows start menu. After starting its main window should open.



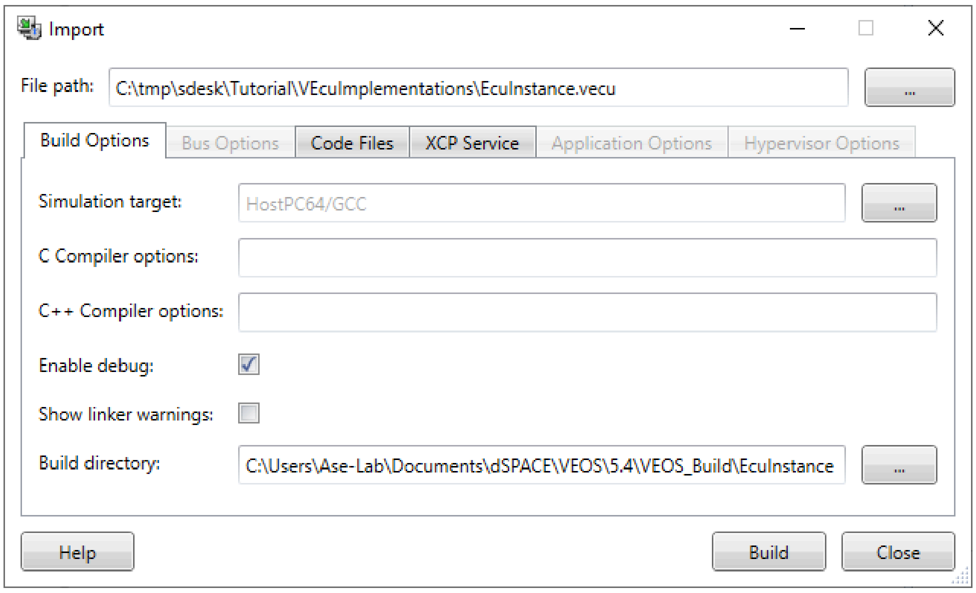


```
Severity   Module       Time           Message
Info       VEOS Player  10:45:00.983   VEOS 5.4 (Version 5.4.0.780265).
  Info     VEOS Player  10:45:07.077   Start completed.
```

Simulation state: Disconnected

3.6.2. Creating a New Simulation Project

In the main ribbon click on “New” to create a new simulation project. Then click on “Import” and select the “EcuInstance.vecu” file which has been created by SystemDesk for your project in the corresponding “VEcuImplementations” folder. The “Import Window” will open. Confirm this by clicking on “Build”.



Import

File path: C:\tmp\sdesk\Tutorial\VEcuImplementations\EcuInstance.vecu

Build Options | Bus Options | Code Files | XCP Service | Application Options | Hypervisor Options

Simulation target: HostPC64/GCC

C Compiler options:

C++ Compiler options:

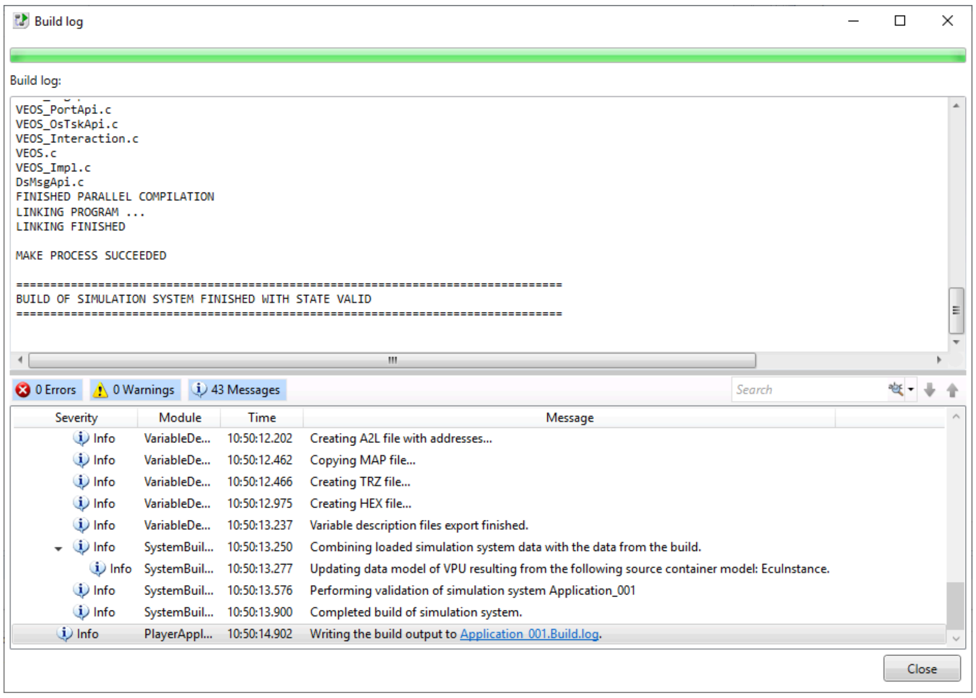
Enable debug: ✓

Show linker warnings: ☐

Build directory: C:\Users\Ase-Lab\Documents\dSPACE\VEOS\5.4\VEOS_Build\EcuInstance

VEOS will now compile and link your AUTOSAR application for the host's PC architecture to allow testing.

Once the build is finished, close the Build log window.



Build log

Build log:

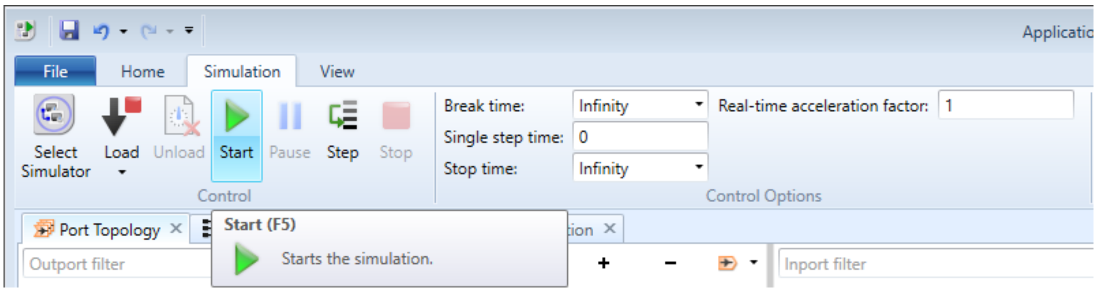
```
VEOS_PortApi.c
VEOS_OsTskApi.c
VEOS_Interaction.c
VEOS.c
VEOS_Impl.c
DsMsgApi.c
FINISHED PARALLEL COMPILATION
LINKING PROGRAM ...
LINKING FINISHED

MAKE PROCESS SUCCEEDED

================================================================================
BUILD OF SIMULATION SYSTEM FINISHED WITH STATE VALID
================================================================================
```

0 Errors | 0 Warnings | 43 Messages

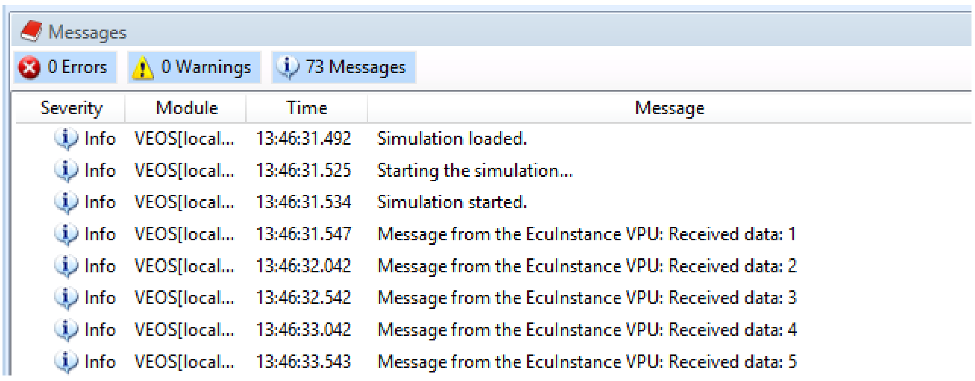
Severity	Module	Time	Message
Info	VariableDe...	10:50:12.202	Creating A2L file with addresses...
Info	VariableDe...	10:50:12.462	Copying MAP file...
Info	VariableDe...	10:50:12.466	Creating TRZ file...
Info	VariableDe...	10:50:12.975	Creating HEX file...
Info	VariableDe...	10:50:13.237	Variable description files export finished.
Info	SystemBuil...	10:50:13.250	Combining loaded simulation system data with the data from the build.
Info	SystemBuil...	10:50:13.277	Updating data model of VPU resulting from the following source container model: EcuInstance.
Info	SystemBuil...	10:50:13.576	Performing validation of simulation system Application_001
Info	SystemBuil...	10:50:13.900	Completed build of simulation system.
Info	PlayerAppl...	10:50:14.902	Writing the build output to Application_001.Build.log.

3.6.3. Running the Simulation

You are now ready to run a simulation of your AUTOSAR application. For this switch to the Simulation tab and click on Start. When asked to save the Veos project file (OSA) do so next to your SystemDesk project file.



If you implemented the task correctly, you should now see output from your DataLogger which prints the data values it receives similar to the following figure:



Severity	Module	Time	Message
Info	VEOS[local...	13:46:31.492	Simulation loaded.
Info	VEOS[local...	13:46:31.525	Starting the simulation...
Info	VEOS[local...	13:46:31.534	Simulation started.
Info	VEOS[local...	13:46:31.547	Message from the EcuInstance VPU: Received data: 1
Info	VEOS[local...	13:46:32.042	Message from the EcuInstance VPU: Received data: 2
Info	VEOS[local...	13:46:32.542	Message from the EcuInstance VPU: Received data: 3
Info	VEOS[local...	13:46:33.042	Message from the EcuInstance VPU: Received data: 4
Info	VEOS[local...	13:46:33.543	Message from the EcuInstance VPU: Received data: 5

Congratulations, you have now finished designing a basic AUTOSAR application which contains 2 Software Components communicating over ports and built and tested it using a virtual ECU. In the final task you will extend your application with a third component which does some very simple data processing.

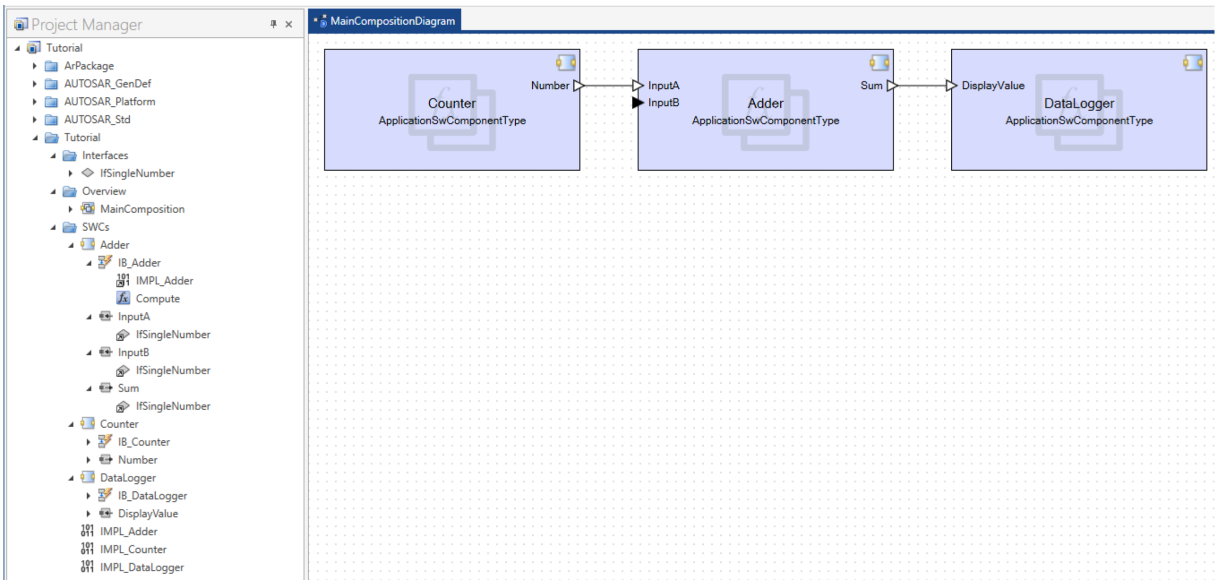
3.7. Practical Task 7: Implementing a Processing SWC

Based on the knowledge and insights you gained in the last tasks, you will now extend your existing AUTOSAR application with a third SWC which will process the data output by the Counter SWC before it is fed to the DataLogger component. This means we will now finalize the architecture as shown in the introduction of Section 3.

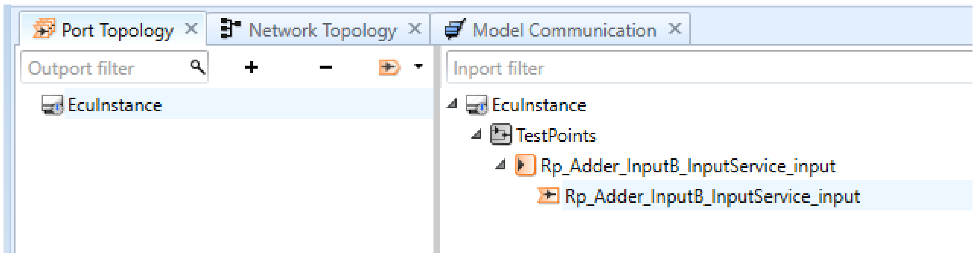
For this, please update your project as follows:

- Create a new SWC called "Adder"
  - Add 2 RPorts named *InputA* and *InputB*
  - Add 1 PPort named *Sum*
  - Assign the interface IfSingleNumber to all ports
  - Add a Runnable named Compute and the corresponding IB and IMPL
  - Correctly set up the data access for the ports
- Create a Data Received Event **for each** RPort so that the Runnable gets triggered whenever the input data changes
- Add an implemention code file named *Adder.c*
  - Implement the C function so that the component reads the data from the 2 RPorts and writes the sum of the two numbers to its PPort
- Update your *MainComposition* so that the Adder SWC sits between the Counter and DataLogger
  - Leave the port *InputB* unconnected

The final project & composition should look like this:



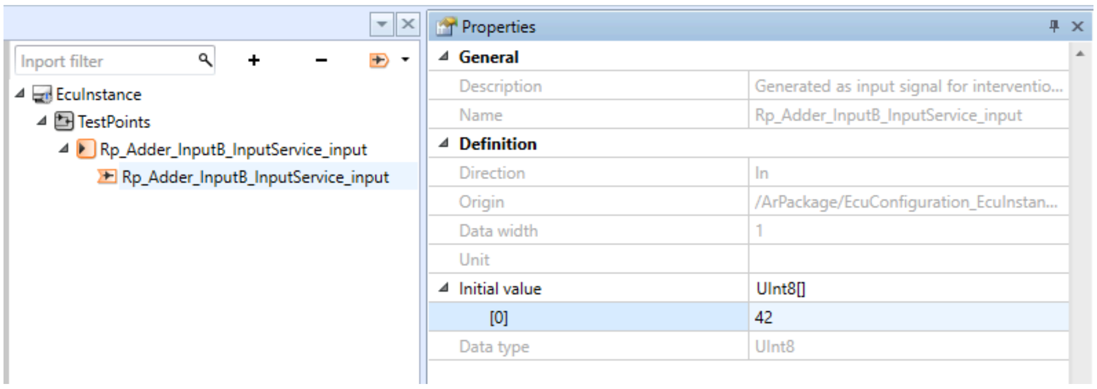
Build a new Virtual ECU (remember to **delete** the old *ArPackage*) and load your application into VEOS. In the **Port Topology** tab, you should see that the port we left unconnected in SystemDesk is now available as test point (expand the EcuInstance nodes to see it).



EcuInstance

- EcuInstance
  - TestPoints
    - Rp_Adder_InputB_InputService_input
      - Rp_Adder_InputB_InputService_input

By using test points you can set the (initial) value for a port directly in VEOS and consequently change the input to the AUTOSAR application manually for testing. To do so, select the test point in the *Port Topology* view and expand the *Initial value* field in the *Properties* window:



- EcuInstance
  - TestPoints
    - Rp_Adder_InputB_InputService_input
      - Rp_Adder_InputB_InputService_input

General	
Description	Generated as input signal for interventio...
Name	Rp_Adder_InputB_InputService_input
Definition	
Direction	In
Origin	/ArPackage/EcuConfiguration_EcuInstan...
Data width	1
Unit	
Initial value	UInt8[]
[0]	42
Data type	UInt8

Enter any (valid) value and re-run the simulation to check that the output is correct based on your expectation.
Show and explain the behavior to the supervisors.

Congratulations, you have finished Unit 3. Based on the workflows you learned, in Unit 4 you will implement a simple ADAS using SystemDesk. However, as manually testing using testpoints in VEOS would be very tedious and time-consuming you will learn how a car simulation can be connected to test behavior more realistically.