ETAS COSYM V3.4.1

vNET Guide

www.etas.com

Copyright

COSYM V3.4.1 I vNET Guide R01 EN - 04.2024

Contents

1 About this Document

1.1 Classification of Safety Messages

Safety messages warn of dangers that can lead to personal injury or damage to property:

 **DANGER**

DANGER indicates a hazardous situation that, if not avoided, will result in death or serious injury.

 **WARNING**

WARNING indicates a hazardous situation that, if not avoided, could result in death or serious injury.

 **CAUTION**

CAUTION indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.

NOTICE

NOTICE indicates a situation that, if not avoided, could result in damage to property.

1.2 Demands on Technical State of the Product

The following special requirements are made to ensure safe operation:

- Take all information on environmental conditions into consideration before setup and operation (see the documentation of your computer, hardware, etc.).

Further information, refer to Safety advice document which is embedded in the COSYM V3.4.1

2 VNET Workflow

The VNET documentation contains the following sections to address individual stakeholders and help them in the integration of VNET:

- A VECU Developer Guide
- A System Integrator Guide
- Troubleshooting Guide

2.1 A VECU Developer Guide

How to integrate VNET in VECU tool-chain?

In this page, we will focus on enabling the VECU Developers to setup a VNET interface to their VECU and test the setup.

VNET stands for Virtual Network.

The following are addressed below:

- How does the network simulation work in COSYM SiL?
- What is the state machine of the VNET simulation?
- How to integrate a VNET interface into VECU tool-chain?

Prerequisites

- Installed COSYM
- COSYM_VX.X.X_vNET Guide where X is the version number

How does the network simulation work in COSYM SiL?

In COSYM SiL, there is an abstraction of communication at signal level and at (bus level) virtual networks.

- Every model imported in COSYM contains three basic processes namely the Init, Exit, and Process.
- In the OS schedule of your COSYM project, you can only describe the signal level communication order.

Fundamentals

An artifact imported in COSYM can contain the data-exchange capability at signal level and at bus level.

We term the data-exchange over signal level as Signal level communication, in which the OS schedule is the UI for the user to order the execution.

We term the data-exchange over bus as Virtual network communication in the case of SiL.

- The network simulation is currently configured using the network configuration xml file.
- This file is called as the `eRunController.xml`, generated after the COSYM SiL Project build phase.
- The `eRunController.xml` is the configuration file that contains both a translation of your OS Schedule and the network configuration definition.
  - The translation of your OS schedule is generated today during the pre-build phase.
  - The network configuration is automatically generated and the User can perform the "Virtual Wiring" or [Bus assignment] during run time in ETAS EE with the help of Bus Configuration UI.

> **Note**
>
> Refer to "General Concepts" on page 22, where you will find an example `eRunController.xml` description with an example system. Refer to this file snippet to understand how your VECU interface must be defined.

2.1.1 Task 1

Create your VECU network interface description

Based on the General Concepts, create a network description file for your VECU.

A Sample `modelControllers.config` has been shown below for a VECU that has two network interface (2 CANs).

- The field **logicalName** can be provided based on the VECU controller definition.
- The field **fullName** is a name which will be shown only to the user in COSYM and this will not be consumed by any software.
- The fields **selfReception** and **busTermination** in the controller properties are configuration settings which can be safely set to false unless the user wants them and has an application in his VECU.

```
{
  "modelName": "vECU_vNET",
  "controllersInfo": [
    {
      "busType": "CAN",
      "controllers": [
        {
          "logicalName": "CAN:1",
          "fullName": "vECU_vNET_VersionName",
          "busProperties": {
            "baudRate": "500000",
            "fastDataMultiplier": "0",
            "stepRate": "1000"
          },
          "controllerProperties": {
            "enableSelfReception": "false",
```

```
            "enableBusTermination": "false"
          }
        },
        {
          "logicalName": "CAN:2",
          "fullName": "vECU_vNET_VersionName",
          "busProperties": {
            "baudRate": "500000",
            "fastDataMultiplier": "0",
            "stepRate": "1000"
          },
          "controllerProperties": {
            "enableSelfReception": "false",
            "enableBusTermination": "false"
          }
        }
      ]
    }
  ]
}
```

2.1.2 Task 2

Include the modelControllers.config file in your VECU build tool-chain and store this information in the resource folder of your FMU[1)] .

Assumption: We assume here that you will generate your VECU as an `*.fmu`.

Once the above information is available, the System integrator can integrate the VECU for his system.

Now, after defining your network, you must configure your VECU to interact with the VNET interface.

> **Note**
>
> Refer to "vNet Basics and example usage with ETAS Virtual Networks" on page 39 to understand the VNET interface APIs.

Let us assume now that based on the above ("vNet Basics and example usage with ETAS Virtual Networks) example, you have adapted your VECU to invoke the APIs.

With start of the simulation in COSYM, first the signal level communication is initialized followed by initialization of the VNET.

The simulation increments following the state machine of VNET for virtual network simulation.

State Machine of VNET Simulation

The state machine of the VNET has the following states :

1) FMUs compliant with version 2.0 of the FMI standard for co-simulation

- Initialization
- Runtime
- Exit

During these above states, a sequence of API calls must be executed by the VECU. These are documented in the section, "Basic API Calls" on page 39.

Integrating VNET Interfaces to VECU

By following the sequence of API calls from the state machine above, you can implement the VNET Interface to your VECU.

Concept of Timing in VNET

Below you will find a sequence diagram of the API calls that are required by a model (VECU/C-Code model) to establish a virtual network simulation. As an example, here we explain for CAN and Automotive Ethernet (AEth).

> **Note**
>
> From COSYM V2.4.0 onwards, the model wrapper handshakes with the simulator and interacts with the model.
>
> Please note that this is only the state representation of VNET. Based on the available states and properties of your model, these states must be ordered.

For example, in some VECUs, the initialization state begins only after an ignition (KL15) to imitate a real ECU behavior. In order to represent the same in simulation, they offer pre-initialization step for the simulation environments.

Therefore in this example, the Initialization state of the VNET will be represented in two parts of the VECU (Pre-initialization step and initialization step) in order to allow the virtual time to increment as in a real vehicle.

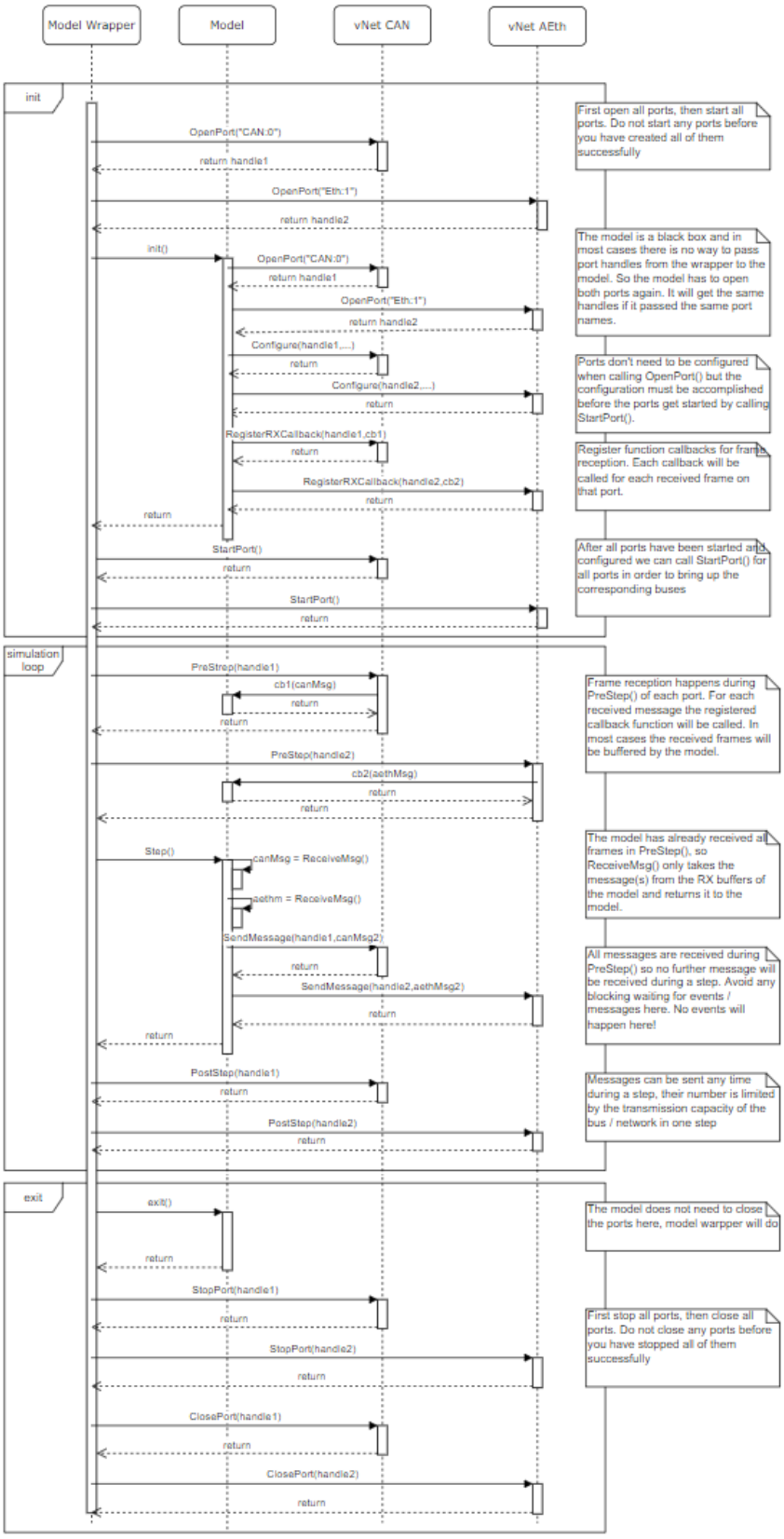


Fig. 2-1: VNET State machine

2.2 A System Integrator Guide

2.2.1 Integrating VECU with VNET and Performing Simulation

In this section, we will focus on enabling the COSYM users to setup a COSYM project with VNET and perform simulation.

The following are addressed:

- How to integrate a VECU that has a VNET interface in COSYM SiL?
- How to create a network interface model with VNET in COSYM SiL for bus protocols?
- How to schedule OS for your System?
- What additional steps are required in the System Integrator Guide : How to integrate VECU with VNET and perform simulation? Read as "Manual Tasks"?
- How to observe the bus communication in EE?.

Refer to the Fig. 2-2 for reference.

2.2.2 System Integrator Workflow in COSYM SiL

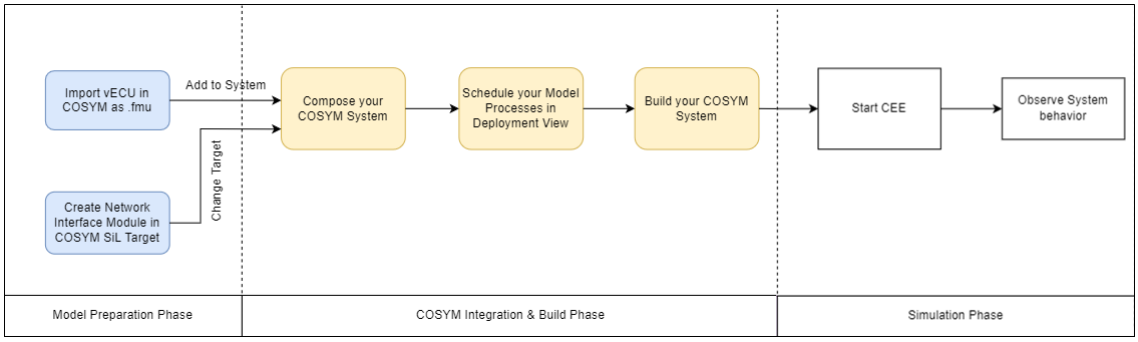


Fig. 2-2: vNETWorkflow

2.2.3 Integrating a VECU that has a VNET interface in COSYM SiL

- The workflow has been tested for VECU packed as an `*.fmu`. The binaries must be available as `*.dll`.

Prerequisites

A. Place the `modelControllers.config` file in the resource folder of VECU
B. The `*.fmu` must be built for FMI 2.0 for co-simulation and the external dependencies of the `*.fmu` must be satisfied.

Integration Step in COSYM

A. Import the `*.fmu` in COSYM
B. Add the `*.fmu` to a system

Scheduling Process for `*.fmu`

In the COSYM, you can schedule the FMU process.

The `*.fmu` is internally responsible for scheduling the bus communication.

> **Note**
>
> — In the VECU, the bus communication time-period and FMU process time-period can be different.
> For example, a VECU can step every 1 ms and the CAN bus in the VECU sends messages every 10 ms.

2.2.4 Creating a C-Code model

1. Right-click on one of the folders in the library and click **Create C-Code** model.
2. Fill out the template Window that opens up to create a COSYM C-Code model for you in the selected folder of the library View.
3. Create a `modelControllers.config` file based on the template provided in the Task 1.
4. Place the `modelControllers.config` file in the source folder of your C-Code model..

> **Note**
>
> Navigate to your COSYM project folder and look inside the library folder for a folder with your C-Code model name. Here, you will find the source folder where you can place the above mentioned file.

Scheduling your C-Code model

1. Open the C-Code module in the model dashboard view.
2. Open the "Process" tab and edit the time period of the Timer process with name "Process".
3. Save the model and map the process to a Timer task in the Deployment View.

Building your COSYM Project

1. Click on **Build and Run** option to enable the Build process of COSYM SiL.

2.2.5 Creating a CAN Network Interface Model with VNET in COSYM SiL for Bus Protocols

— The workflow has been tested for CAN/CAN-FD protocols using the NIC model in COSYM SiL.

Prerequisites

- COSYM NIC license.
- CAN `configuration.xml` or a DBC file defining the CAN/CAN-FD network messages.

Create a CAN Network Model in COSYM

- Follow the steps mentioned in the `CAN Editor User Guide.pdf` shipped along with the COSYM installer. After installing COSYM, you will find this document here: <COSYM_Installation_Directory>`\Documentation\Editors` folder.

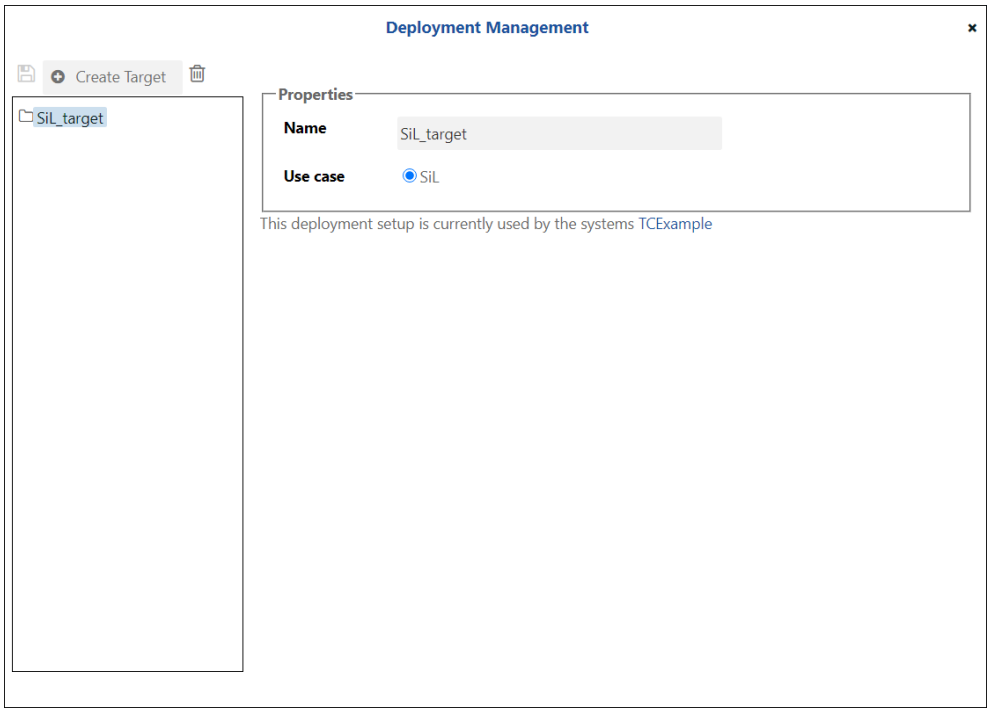


Fig. 2-3: The "Deployment Management" dialog box

Scheduling your NIC Model

- Use the COSYM "Auto mapping" feature of COSYM to automap NIC process to Tasks.
- Delete the "BusLoadMonitoring" processes from the Task lists.
- If you wish to map manually, pack "Send" and "Receive" processes within a single Task. The Task time can be set manually by you.

Building your COSYM Project

1. Click on **Build and Run** option to enable the Build process of COSYM SiL.

2.2.6 Observing Simulation in ETAS EE

> **Note**
>
> Before observing the bus communication, observe the simulation start and time increment in COSYM.

Configuring Bus and Virtual Wiring

1. Click on the **Bus Config** icon on the ETAS EE Toolbar as shown below. All the models which have a virtual bus configured, will be displayed in the Config. Window with Bus properties.

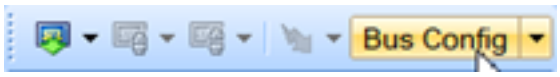


Fig. 2-4: Bus config feature

2. Click on each network interface to assign the right bus based on your system network configuration.

This step in SiL is called as "Virtual Wiring" or "Controller Mapping". This view is shown below:

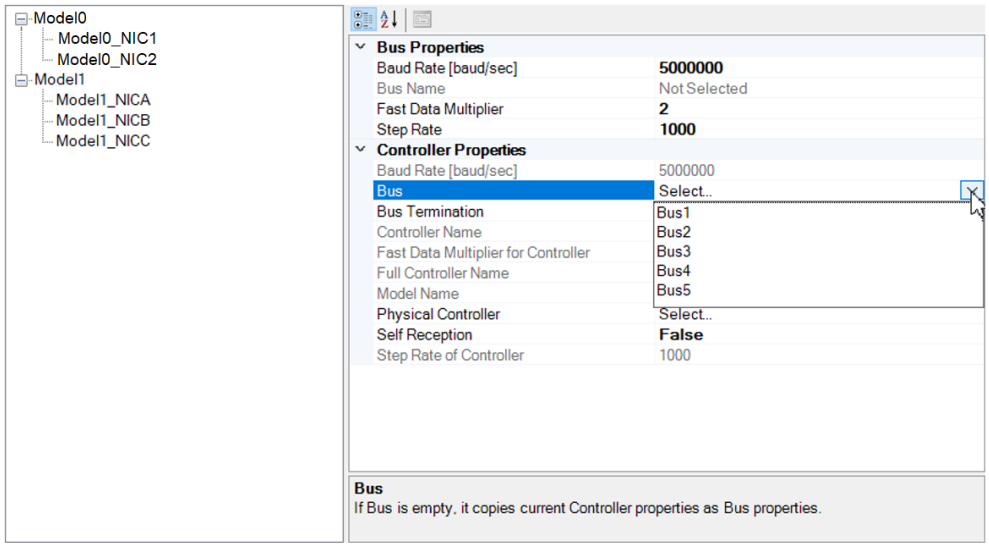


Fig. 2-5: Bus and controller mapping

As shown in the figure above, when you click on the drop down menu, you will observe a list of bus to which your network interface can be connected.

3. Select the bus appropriately based on your network configuration.
4. Click on **Save**, in order to preserve this information for your simulation.
5. You can now proceed to download as shown below and prepare your ETAS EE for simulation.

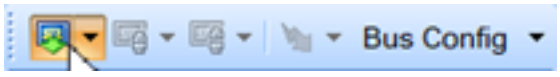


Fig. 2-6: Download

6. Click on **Start** simulation to start the Initilization of VNET.
7. Observe whether the simulation time is incrementing in COSYM.
8. Check the values of the messages in EE for the signals of interest.

2.3 Troubleshooting Guide

COSYM SiL Extended Functionalities

In this page, we will focus on applying the available troubleshooting methods for COSYM SiL and understand the underlying network and simulation behavior.

We will address the following:

- What should I do when my simulation does not step?
- What information does the troubleshooting steps provide me to identify my problem?

Prerequisities

COSYM V3.4.1 version installed and ESSE version 1.3.2.0 or higher version on your PC.

> **Note**
>
> By enabling the below settings, please be aware that your simulation speed will be drastically reduced.

Generating Performance Reports

Using the below mentioned environment variables, you are able to measure the performance of computational for a running COSYM project.

When do you need this performance reports?

If you observe that your simulation speed is not optimal or not as fast as possible and would like to analyze what is causing your system to slow down.

1. Go to <COSYM_installation_location>`\Simulator\win64\atssimulator\scripts\` folder and edit the `eRunController.bat` file in any of the text editor.
2. Add the following line of code in the `eRunController.bat` file as shown in the Fig. 2-7.

eRunController.bat

```
set ESSE_reportFile=%DEPLOY_DIRECTORY%\ModelReport
```

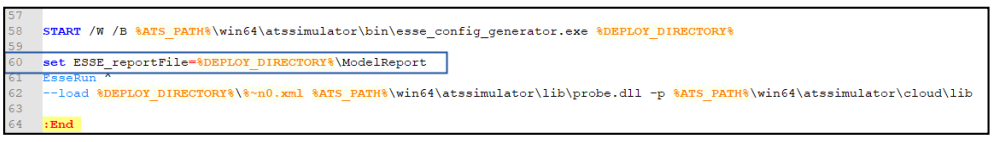


Fig. 2-7: Adding code to generate a report

By adding the line of code, COSYM generates the reports for each artifact inside your temp folder (folder name with project id) for individual models.

What does the performance reports indicate?

From the performance reports, following can be understood:

- The slowest model in the system.
- Busy and idle time of every model in the system.

Through this, the User can take action on how the artifact could be improved if an artifact is the slowest model or request support of COSYM experts when Probe is the slowest model in the system.

Observe network traffic and Messages

In order to observe the VNET messages and understand the state machine execution of the simulation, make the following changes in the `eRun-Controller.bat` file as shown in the Fig. 2-7.

1. Add the following lines of code.

eRunController.bat

```
set ESSE_traceWithSeq=1

set ESSE_disableInitialEsseReport=0

set ESSE_enableInitialEsseReportSnf=1

set ESSE_enableInitialEsseReportSnfWait=1
```

2. Uncomment the following line of code.

```
rem set ESSE_traceFile=%DEPLOY_DIRECTORY%\ModelTrace
```

```
54  set ESSE_traceWithSeq=1
55  set ESSE_disableInitialEsseReport=0
56  set ESSE_enableInitialEsseReportSnf=1
57  set ESSE_enableInitialEsseReportSnfWait=1
58  set ESSE_traceFile=%DEPLOY_DIRECTORY%\ModelTrace
59  rem ESSE Report Shared Memory Size set to 1MB
60  set ESSE_shmReportBlockSize=1048576
61
62  START /W /B %ATS_PATH%\win64\atssimulator\bin\esse_config_generator.exe %DEPLOY_DIRECTORY%
63
64  set ESSE_reportFile=%DEPLOY_DIRECTORY%\ModelReport
65  EsseRun ^
66  --load %DEPLOY_DIRECTORY%\%~n0.xml %ATS_PATH%\win64\atssimulator\lib\probe.dll -p %ATS_PATH%\win64\atssimulator\cloud\lib
```

Fig. 2-8: Adding and uncommenting lines of code

These reports are generated in the temp folder on your PC. In the temp folder, you can find a folder name with project id inside which `xxx.esse`Trace files for individual models are created.

> **Note**
>
> You can also change the path of the trace\report file by setting the environment variable (ESSE_traceFile, ESSE_reportFile) to <destination_directory>\<filename>.

What does the ESSE reports indicate?

- The initialization values of the configuration.
- The step execution log of every model in the system.
- In the case of VECU and NIC module, the state machine of vNET
  - e.g. start simulation, Open VNET port, PreStep function call, postStep function call.
  - Messages Send and Receive action performed by the VECU with an address.
  - Exit function and closing of VNET port.
  - Error messages if the VNET API received any exception.
  - Error messages if the simulator could not step the module.

3 General information on COSYM vNet

The COSYM VNET is based on the ESSE Systems Engineering Workbench for ETAS that enables a structural architecture for the design of network- connected, heterogeneous plant and control models, to be accurately modeled and simulated with near maximum performance on multi-core computers.

The ESSE Distributed Simulation Manager (DSM) and ESSE Distributed Simulation Engine (DSE) are an integral part of the workbench.

Together, they form a high performance, scalable, completely distributed, multi-core capability for managing the simulation of small and large scale, distributed systems and systems of systems models.

The DSM initiates the distributed simulation by commands from a simulation launch script and propagates Start/Stop and Pause/Resume commands to all compatible simulating processes.

The DSE initializes each modeled communication network and manages the synchronization of processes and multiple communications between them so that near optimal overlap of process execution is achieved.

ESSE System Network Fabrics (SNF networks) are high performance, timing accurate models of signal communication, control and network protocols such as CAN, CAN- FD, FlexRay and Automotive Ethernet.

SNF network models in conjunction with the DSE manage the synchronization and data exchange between distributed models.

ESSE System Network Fabric SyncSignals (SnfSyncSignals) acts as a distributed communication model that enables signals of various types to be communicated between models in a distributed model. It also provides the means whereby signals are communicated and synchronized with third party tools.

COSYM VNET integrates the ESSE components in one single library with an API which is easy to understand and can be used directly by C/C++ code.

3.1 Network Protocol Modeling Capabilities

A distributed system model can be constructed from any number of models (including any number of coupled, third party tool based, models).

Any number of each type of SNF network can be instantiated within a distributed system model.

Each instance of an SNF network can have any number of connections to models within the system.

These numbers are limited only by the available simulation system resources.

Each SNF network instance provides time synchronization with its connected models.

A master controller model provides synchronization between the Experiment Environment and the distributed system simulation.

Each SNF network instance accurately models the arbitration and transmission times associated with the particular network type and configured topology.

Each SNF network is capable under full load of better than real-time performance.

The overall simulation performance is, however, limited by the performance of the slowest simulation model within the overall distributed system simulation.

4 ETAS Virtual Networks (VNET)

This chapter contains the following sections.

4.1 General Concepts

- A vNet simulation consists of at least one network configuration file and some simulation binaries (executable files).
- The network configuration is represented by an XML file.
- The simulation is based on virtual networks and its interfaces as given below:
  - No real network interfaces are required.
  - Virtual network interfaces are not visible to the operating system and standard applications.
  - It is not possible for standard network applications like e.g. `tcpdump` to use virtual networks.
  - However, it is possible to capture network traffic on virtual vNet interfaces and forward it to a real network interfaces.

4.2 Basic Requirements to use Virtual Networks

- All simulation binaries which you want to communicate via ETAS Virtual Networks must be running as application processes on the same machine (Linux or Windows)
- Several models can be in the same binary executabale file and share the same process
- All applications must link to ESSE libraries
- Run `EsseRun.exe` to launch all applications which is available at <COSYM installation path>`\Simulator\`<operating system>`\esse\develop\bin64` folder.

- All applications must use the vNet API or the SNF API to connect to the virtual networks
- An XML configuration file must exists. Additional configuration files might be necessary for special configurations. For example, `SyncSignalNetworks` which are not described in this document

4.3 Global Configuration

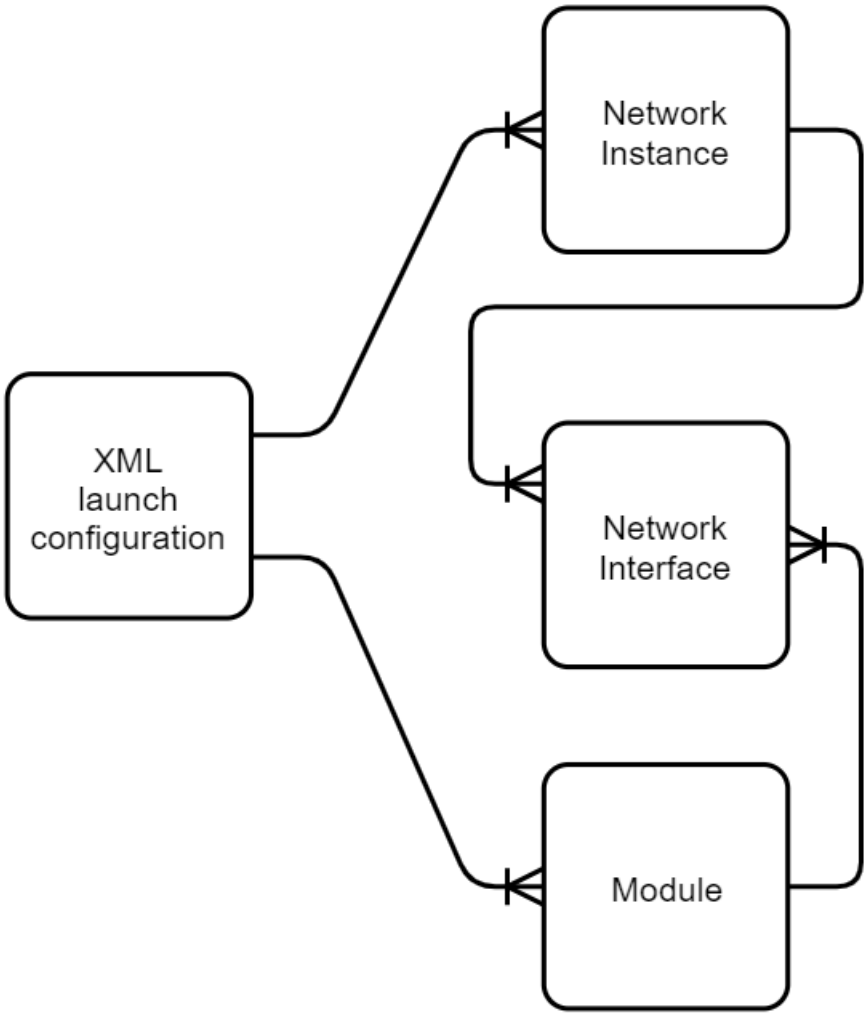
A single global configuration file is required to configure the network simulation, it is used by `EsseRun` to start the simulation. This file contains the description of the network instances and their parameters with the tag `SnfInstance`.

- It is evaluated by `EsseRun` to start all simulation applications using ETAS Virtual Networks in a synchronized way.
- It describes the global topology with all networks and network interfaces.
- It provides configuration information for all networks and network interfaces; For example, `baud rates`.
- It provides a configuration for simulation models; For example, `stepping rates`.
- The configuration is static in the sense that all information must be present in advance, there is no possibility to add additional models, interfaces and networks while the simulation is running, there is no hot-plugging.

The EsseRun configuration is an XML file and contains the following information:

- A set of network instances, for each network instance the following information is needed:
  - Network type
  - Unique name
  - Configuration parameters, for example, the `stepping rate` of the network
- All applications (also called models) to be started for the network simulation. For each application, it is necessary to have:
  - An unique name
  - Simulator type (e. g. EsseTest)
  - A shell command to launch the application
  - A set of network interfaces, for each interface we need
    - the name of the network it is connected to
    - an alias name which is actually the name of the network interface port (e.g. "eth0" or "CAN1"), this alias name is a local name of the network interface and doesn't need to be globally unique
- Note that any information can be provided in the form of Environment Variables, the values of which will replace placeholders $(<variable name>) before the content is evaluated.

The data model of the configuration files in terms of entities and relations between those entities is described by the following entity relationship diagram:



- Each configuration file is composed of many models and networks
- Each model can be connected to many network interfaces
- Each network instance can have many network interfaces connected to it
- All entities may have additional configuration data

4.4 Available Network Types

ESSE offers implementations for various network technologies common in automotive application area:

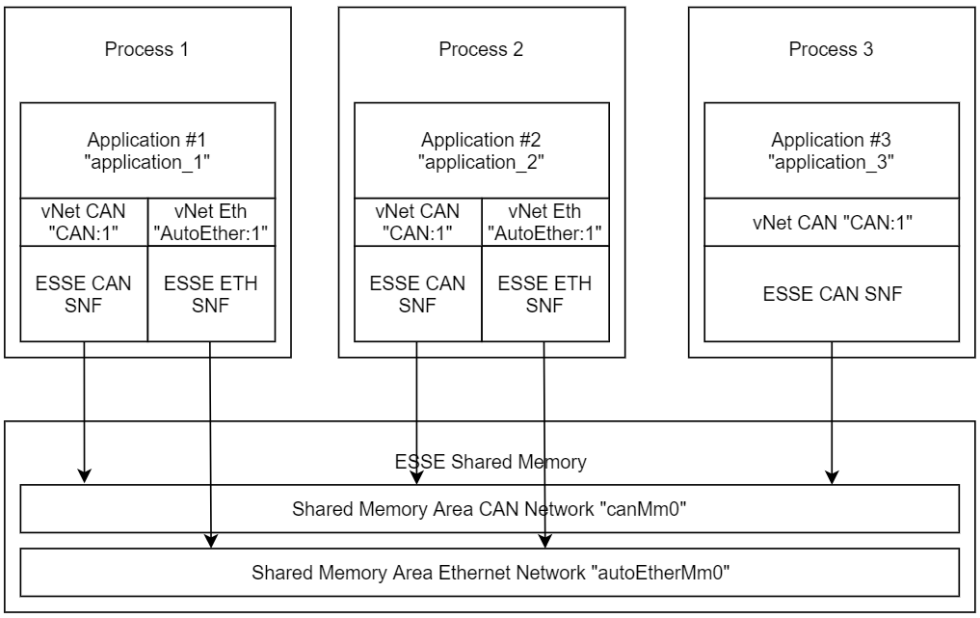
Network Type	Internal Name	Comment
Direct signal connection	SnfSyncSignals	This network type is not described in this document. This is not a network but actually a direct wired connection between models. One instance of `SnfSyncSignals` and one instance of `SnfControl` (not mentioned in this list) are utilized by COSYM / EE for communication of signals and the run-time control of a complete system simulation.
Automotive Ethernet	SnfAutoEtherMm	Automotive Ethernet implementation available with speeds 1 Gigabit and 100 Megabit
CAN (FD)	SnfCanMm	CAN 2.0 and CAN FD implementations with both identifier lengths: 11 and 29 bits
FlexRay	SnfFlexRayMm	FlexRay implementation
LIN	SnfLin	LIN implementation

4.5 Example

In the following example, we consider 3 applications running as 3 processes are communicating via 2 different networks such as CAN and Ethernet.

All 3 applications use the CAN network, but only the applications 1 and 2 are connected to the Ethernet network.

So 5 network interfaces in total need to be created and configured:



A corresponding EsseRun configuration file might look as follows:

```
<EsseLaunchConfig projectName="example">

    <SnfInstance
        snfInstanceName="autoEtherMm0"
        snfTypeName="SnfAutoEtherMm"
        snfConfigString="$(configEth)"
    />
    <SnfInstance
        snfInstanceName="canMm0"
        snfTypeName="SnfCanMm"
        snfConfigString="$(configCan)"
    />

    <ModuleInstance
        moduleInstanceName="application_1"
        moduleSimulatorType="EsseTest"
        commandString="$(command1)"
        >
        <SnfConnection
            snfInstanceName="autoEtherMm0"
            snfAliasName="AutoEther:1"
        />
        <SnfConnection
            snfInstanceName="canMm0"
            snfAliasName="CAN:1"
        />
    </ModuleInstance>

    <ModuleInstance
```

```
        moduleInstanceName="application_2"
        moduleSimulatorType="EsseTest"
        commandString="$(command2)"
        >
        <SnfConnection
            snfInstanceName="autoEtherMm0"
            snfAliasName="AutoEther:1"
        />
        <SnfConnection
            snfInstanceName="canMm0"
            snfAliasName="CAN:1"
        />
    </ModuleInstance>

    <ModuleInstance
        moduleInstanceName="application_3"
        moduleSimulatorType="EsseTest"
        commandString="$(command3)"
        >
        <SnfConnection
            snfInstanceName="canMm0"
            snfAliasName="CAN:1"
        />
    </ModuleInstance>
</EsseLaunchConfig>
```

According to the above description the configuration contains the following entities:

- Network instance "autoEtherMm0" of type Automotive Ethernet
- Network instance "canMm0" of type CAN
- Application "application_1" with configuration and two interfaces
  - "AutoEther:1" to network instance "autoEtherMm0"
  - "CAN:1" to network instance "canMm0"
- Application "application_2" with configuration and two interfaces
  - "AutoEther:1" to network instance "autoEtherMm0"
  - "CAN:1" to network instance "canMm0"
- Application "application_3" with configuration and one interface
  - "CAN:1" to network instance "canMm0"

Note

- The alias names snfAliasName - they define network interface names (port names).
- The network interface names do not need to be globally unique, they only need to be unique within each model.
- The entities in the XML file refer to each other by using their unique names. These references are represented by arrows in the entity relationship diagram shown above.

4.5.1 Place-holders

Most of the configuration is done by providing place-holders which will be replaced by values of environment variables.

- $(configEth) shall be replaced by an ethernet configuration string composed of key-value-pairs separated by commas and describing details of the Ethernet network
- $(configCan) shall be replaced by a CAN configuration string composed of key-value-pairs as described above
- $(command1) shall be replaced by a shell command invocation string including all command line arguments for application 1
- $(command2) shall be replaced by a shell command invocation string including all command line arguments for application 2
- $(command3) shall be replaced by a shell command invocation string including all command line arguments for application 3

Due to the environment variables to be set the best way to launch `EsseRun` is via a batch file. A typical file to start 3 processes on a Windows machine might look like the following one:

Batch file for EsseRun

```
"%REBUS_ESSE_BIN_DIR%\EsseRun.exe" ^
--setEnv command1 "cmd.exe /K start %BASE_DIR%\application_1.exe arg1
arg2" ^
--setEnv command2 "cmd.exe /K start %BASE_DIR%\application_2.exe arg1"
^
--setEnv command3 "%BASE_DIR%\application_3.exe" ^
--setEnv configEth "ticks=1G,stepTicks=1M,busTicks=1" ^
--setEnv configCan "ticks=1M,stepTicks=1000,busTicks=2" ^
--launch %~d0%~p0\example.xml
```

In the example above all environment variables are provided as command line options to EsseRun. The last option is the name of the XML configuration file. EsseRun has no order-dependent arguments, only options.

Note that the command strings are OS-specific and contain all arguments required to launch the individual applications on the specific operating system.

4.6 Stepping Parameters

In the aforementioned example also note the stepping parameters for both networks, Ethernet and CAN:

- ticks
  Number of base ticks per second. Base ticks is a common time unit to describe other timing parameters. The number of base ticks per second must be sufficiently high in order to describe other parameters by integer numbers of base ticks. The value 1G is an abbreviation for 1E9 = 1*10^9 = 1000000000, similarly 1M is 1E6 and 1K is 1E3
- stepTicks

Integer number of base ticks used for 1 simulation step

- busTicks
  Integer number of baseTicks which represent the length of 1 bus cycle

> **Note**
>
> All of the parameters were intentionally chosen to be integers in order to avoid floating point arithmetic. The division operations below shall be computable as integer divisions without rest.

In the example above, the Ethernet configuration in `configEth` is as follows:

- baseTicks = 1000000000
- stepTicks = 1000000
- busTicks = 1

- baseTicks / stepTicks = 1000000000 / 1000000 = 1000 stepTicks per second - the Ethernet network has to be stepped once in every millisecond
- baseTicks / busTicks = 1000000000 / 1 = 1000000000 busTicks per second, so the Ethernet has an internal transmission frequency of 1Gbit/s

The CAN configuration in `configCan` is as follows:

- baseTicks = 1000000
- stepTicks = 1000
- busTicks = 2

- baseTicks / stepTicks = 1000000 / 1000 = 1000 stepTicks per second - the CAN network has to be stepped once every millisecond
- baseTicks / busTicks = 1000000 / 2 = 500000 busTicks per second, so the CAN has an internal transmission frequency of 500 Kbit/s

4.7 Applications

In the example above, 3 different applications (models) would be launched by `EsseRun` (on a Windows machine):

- `application1.exe` with 2 command line arguments running in a new console window
- `application2.exe` with 1 command line arguments running in a new console window
- `application3.exe` without command line arguments running in the original console window

The requirements that must be fulfilled by the applications in order to connect to virtual networks are:

- Actually, any application can be started in this way, it might even have a GUI and require user interaction
- The application has to connect to an ESSE vNet network interface; it has to configure, open and start the interface before any network communication can happen
- It has to step the network interface(s) according to the stepping rate of the corresponding network(s) by calling `preStep()` and `postStep()`. See the next sub-section for further details
- At the end of the simulation it has to stop and close all network interfaces in order to free their resources
- All individual runtime tasks are described "vNet Basics and example usage with ETAS Virtual Networks" on page 39

4.8 PreStep and PostStep

Virtual networks need to be stepped in a similar way like normal simulation models. Normal models have typically a `step()` function which is called at certain points of time during simulation and which is given the step duration as argument.

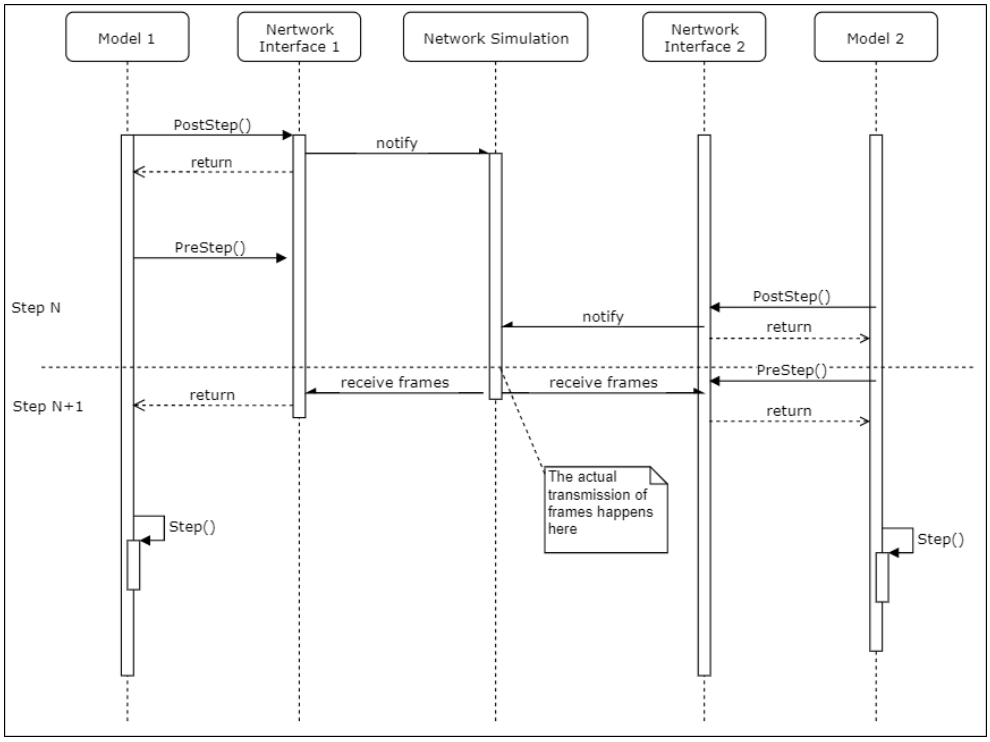
Virtual networks are stepped by two function calls `PreStep()` and `PostStep()`. The reason why two function calls are necessary is that in a model connected to several networks interfaces all interfaces need to be pre-stepped before any of them can be post-stepped. So `PreStep()` and `PostStep()` cannot be assumed in one function call.

`PreStep()` and `PostStep()` together can be understood easily when we look at `PostStep()` in step N and `PreStep()` in subsequent step N+1:

- `PostStep()` tells the network simulation: *I'm done with my network communication for the current step N, I have sent all my frames and no communication will happen until next step*
- `PreStep()` tells the network simulation: *Wait until all other models are done with the current step N and begin the new step N+1 in which I expect to receive new frames in my network interfaces*

So, when we say `PostStep`, we actually say: `notify the others that I'm done`.

When we say `PreStep`, we actually say: `wait for the others and give me the frames.`



Remarks

- Both models finish step N by calling `postStep()`
- Since model 1 is a bit faster, it starts next step N+1 by calling `preStep()` before Model 2 calls `postStep()` for step N
- So Model 1 has to wait - this is achieved by blocking the `preStep()` function call until Model 2 has called `postStep()`
- Once both models are done with step N and have called `postStep()` the ESSE network simulation computes the network traffic and sends it to the hosts
- After that the blocking calls of preStep return and both models can continue with step N+1

4.9 Stepping the Virtual Network Interfaces

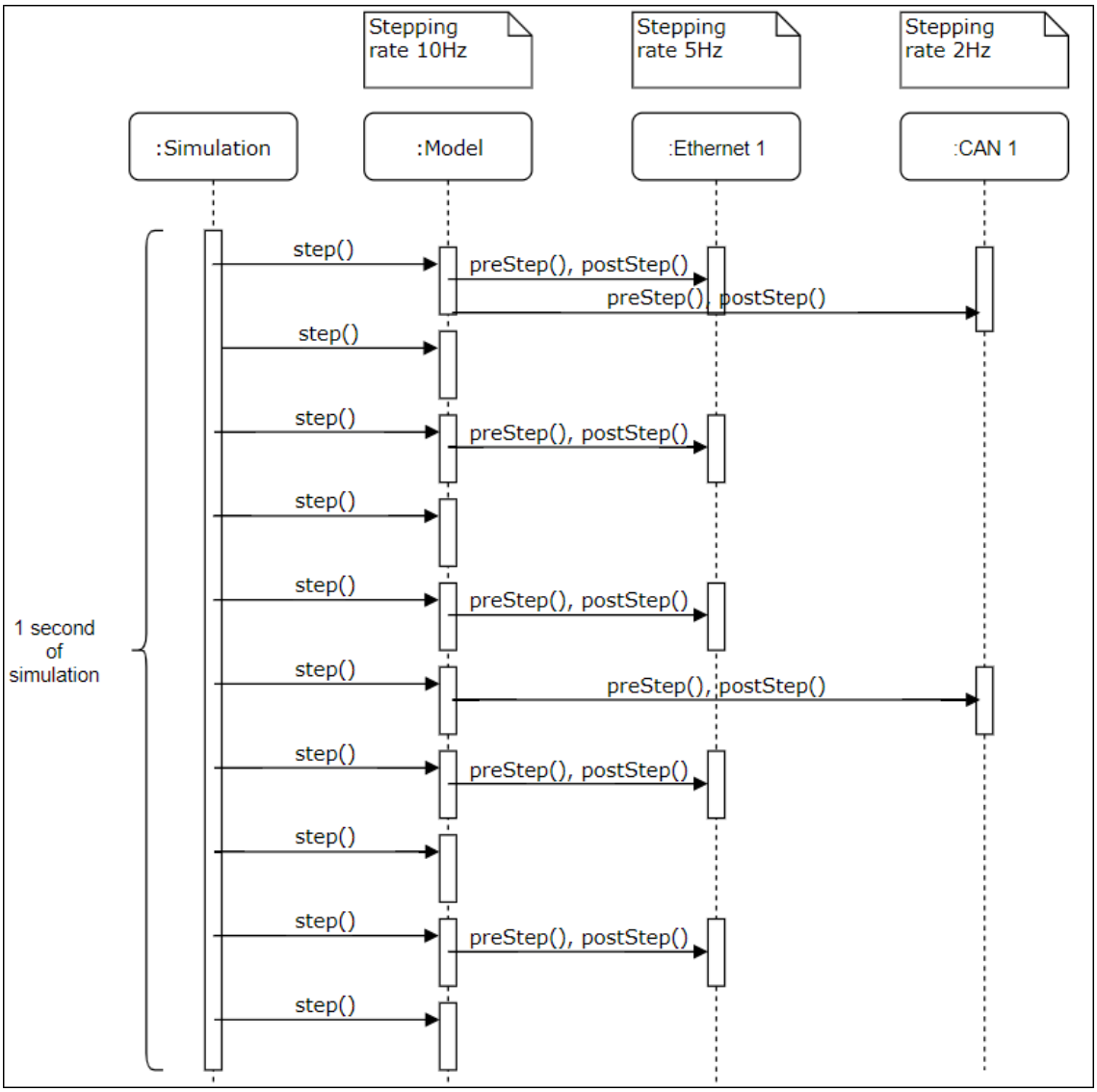
Similar to simulation models, vNet network interfaces need to be stepped in order to make the network communication happen. Each network interface is represented by a tag SnfConnection in the configuartion file and has an alias name which can be queried by the application.

- As shown above all networks need a configuration in the configuration file to determine the stepping rate of the network
- All network interfaces connected to a network share the same stepping rate - that one of the network
- Each network might have its own individual stepping rate, e. g. two CAN networks with the same baud rate may have different stepping rates

- So in each simulation model all network interfaces need to be stepped according to the stepping rates of the networks they are connected to
- The stepping rates of the networks might differ from the stepping rates of the models they are used in

The sequence chart below demonstrates how to step two virtual network interfaces with different stepping rates from a model which also has a different step rate. The step rate of the model must be at least as high as the step rates of the interfaces.

- Model 1 has a stepping rate of 10 Hz
- The network interface Ethernet 1 has a step rate of 5 Hz
- The interface CAN 1 has the lowest step rate of 2 Hz



Remarks

- The main simulation loop just steps the model and doesn't care about which network interfaces the model is using
- So in the sequence chart it is the responsibility of the model to step its network interfaces with the correct rates in the right simulation cycles
- As an alternative, it would also be possible to step the network interfaces directly from the outer main loop
- Unlike the model's step function the ESSE network interfaces require two function calls: `PreStep()` and `PostStep()`
- PreStep() must be called before frames from the corresponding network can be processed
- Between `PreStep()` and `PostStep()` the model's code shall be executed: received frames hall pe processed and new frames shall be sent
- `PostStep()` must be called after frames have been processed and new frames might have been sent to the network the interface is connected to

Note

When embedded in a COSYM/EE environment, model stepping must occur at a rate matching or exceeding the rate at which the model's connected networks are being stepped, otherwise deadlocks may occur.

4.10 Transmission of Frames

The stepping of ETAS Virtual Network types has the consequence that frames are always received at earliest in the next simulation step: A frame send in simulation step N always arrives at the destination interface(s) in simulation step >= N+1.

In principle frames cannot be sent and received in the same simulation step - in-step responses are not possible. This fact also leads to the conclusion that the transmission latency of frames is at minimum equal to the duration of a step: A network with a stepping rate of 1 kHz (1ms step duration) has thus a minimum latency of 1ms.

> **Note**
>
> The actual message transmission latency reported within the received message can be less than a step, as the message arbitration accurately models the particular protocol as if the request for transmission on the wire occurred at the beginning of step N.

4.11 Technical Basics

ETAS Virtual Networks are based on shared memory, they are passive and decentralized:

- There is no *central application* simulating the network itself
- There is no *governor* like a central *network server* which would have a global view to the network(s)
- Each communication partner (each station) is represented by a process that is using a portion of shared memory to communicate with other partners in the same network
- The shared memory serves as a *means of inter-process communication* to exchange data (frames) and synchronize all communication partners which each other
  - The exchange of frames uses virtual memory buffers in shared memory
  - The synchronization between communication partners is realized using *spin locks* blocking on *shared memory variables*

5 VNET

vNet is intended to be a vendor independent API to create and communicate with ESSE or Third Party virtual network interfaces. When VNET is used with ETAS Virtual Networks, exists as an API on top of the ESSE SNF API.

The actual libraries are called ESSE libraries, but vNet is becoming a common name for the interface and the ESSE libraries.

This chapter aims at providing detailed descriptions of all vNet Network Interfaces.

- "vNet Basics and example usage with ETAS Virtual Networks" on page 39
  Describes basics and common facts of all the network types.
- "VNET CAN" on page 55
  Describes CAN network types in detail.
- "vNet FlexRay" on page 70
  Describes FlexRay network types in detail.
- "vNet Ethernet" on page 62
  Describes Ethernet network types in detail.

API

The API is written in two programming languages:

- C language
  The essential parts and the linkage is written in C.
- C++ language
  - On top of the C parts, there are C++ parts which are visible to C++ compilers only and extend the basic functionality in C.
  - The C++ parts are header-only, no C++ linker symbols are generated.
  - The C++ parts make use of the C interface.

The image below shows an example on how vNet can be used by an application to communicate over a virtual network:

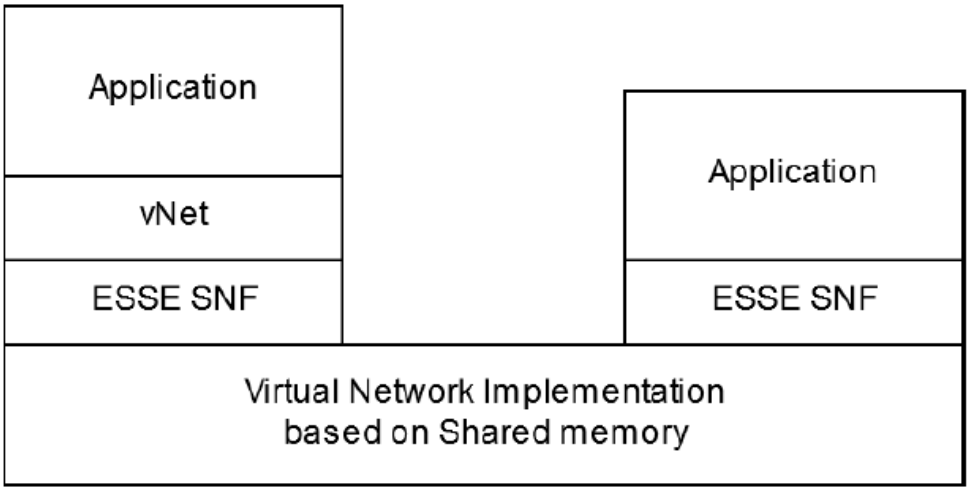


Fig. 5-1: Communication via virtual network

> **Note**
>
> vNet is an abstraction layer and it is also possible to use the SNF API directly. This documentation is only about the vNet interfaces and recommends to use only these interfaces.

5.1 vNet Basics and example usage with ETAS Virtual Networks

5.1.1 Overview

- Although all vNet interfaces (CAN, FlexRay, Ethernet) look similar, they are totally independent from each other
- So there is no common base class
- But the general sequence of API calls is always similar

The following description contains facts common for all vNet interfaces: CAN, Automotive Ethernet and FlexRay. Differences between individual vNet implementations are outlined whenever necessary. Links to pages with the network-specific descriptions are also provided in such cases.

5.1.2 Basic API Calls

The following basic API calls must be done by all applications during initialization, during stepping at runtime and after the simulation. Note that, too many or too few are incorrect calls of `mPreStepPort()` and `mPostStepPort()` which lead to deadlocks or to different virtual times in models. So it is very important to call `mPreStepPort()` and `mPostStepPort()` for each network interface used. Both calls must happen with the correct stepping rate configured for the corresponding network.

Refer to the "Timing diagram" on page 54 for more details.

Initialization

A. Open the shared library `vNetInterfaceSnf.dll` or `libvNetInterfaceSnf.so` (dynamic linking at runtime)
B. Import the symbol of the C function to create the interface object
C. Call the C function to create the vNet interface object
D. Optionally call `mGetAllPortNames()` to get a list of all port names configured for this executable model by network type, and, choose the right port name for the own interface. Alternatively, a fixed port name can be used and it is the responsibility of the configuration to provide an appropriate connection port
E. Call `mOpenPort()` to open the desired port with the chosen name and a given port configuration, a port handle will be returned
F. Call `mStartPort()` with the returned port handle, possibly configure additional properties of the interface

Runtime

A. Call `mPreStepPort()` using the port handle. When all applications have called `mPreStepPort()` for the first time, the network simulation knows that all applications have finished the Initialization and are ready for the cyclic operating mode.

B. During the call of `mPreStepPort()` callback functions will be called to receive frames arrived on the interface.

C. Send frames to the interface using `mSendMessage()`.

D. Call `mPostStepPort()` to finalize the step.

Exit

A. Call `mStopPort()` to terminate the operation of the network interface.

B. Call `mClosePort()` to close the port.

C. Close the shared library handle to unlink from it.

5.1.3 Loading Libraries

- Initial steps aim at loading the shared library and are OS dependent - the technique used here is *dynamic loading and linking of a shared library* at run-time
- The library to load is the vNet shared library `vNetInterfaceSnf.dll` (or `libvNetInterfaceSnf.so` on Unix-like systems)
- The same library is used for all supported vNet interfaces
  - CAN
  - Automotive Ethernet
  - FlexRay

The following example code loads shared libraries in an OS-independent way:

Loading Dynamic Library

```
ifdef VNET_BUILD_WIN

    HMODULE openSharedLibrary(const char* const sharedLibName, int
errorBufferSize, char* errorMsg)
    {

        HMODULE result = LoadLibraryEx(sharedLibName, NULL, LOAD_WITH_
ALTERED_SEARCH_PATH);

        if (result == 0)
        {
            if (errorMsg != NULL)
            {
                LPTSTR pszMessage;
                const DWORD dwLastError = GetLastError();
                FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_
MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
                              NULL,
                              dwLastError,
                              MAKELANGID(LANG_NEUTRAL, SUBLANG_
DEFAULT),
                              (LPTSTR)&pszMessage,
                              0, NULL );

                int c = snprintf(errorMsg,
errorBufferSize, "vNetCanHandle failed to load %s error message is
'%s'\n", sharedLibName, ((pszMessage!=0)?pszMessage:"?"));
                if (c < 0)
```

```
                    errorMsg[0] = 0;
                }
            }
            return result;
        }

        FARPROC getSharedLibrarySymbol(HMODULE module, const char* const
symbolName) {return GetProcAddress(module, symbolName);}
        int closeSharedLibrary(HMODULE module) {return FreeLibrary
(module);}

#else

        typedef const void* FARPROC;
        typedef void * HMODULE;

        HMODULE openSharedLibrary(const char* const sharedLibName, int
errorBufferSize, char* errorMsg)
        {
            dlerror();
            void * result = dlopen(sharedLibName, RTLD_LAZY | RTLD_
GLOBAL);

            if (result == NULL)
            {
                if (errorMsg != NULL)
                {
                    snprintf(errorMsg, errorBufferSize - 1, "vNetCanHandle
failed to load %s error message is '%s'", sharedLibName, dlerror());
                }
                return NULL;
            }
            return result;
        }

        FARPROC getSharedLibrarySymbol(HMODULE module, const char* const
symbolName) {return dlsym(module, symbolName);}
        int closeSharedLibrary(HMODULE module) {return dlclose(module);}

#endif
```

As mentioned here there are different SNF shared libraries for all 3 network types. They will be loaded automatically by the vNet library.

5.1.4 Getting Symbols

- The next step is to import a pointer to a function to create the desired vNet interface, the names of the functions are listed in the 2nd column in the following table
- The function pointer is obtained by importing a link symbol from the shared library
- The import requires the name of the exported C function from the library

Network Type	Header File	Link Symbol Name	Signature f the C Function	Arguments to call the Function
CAN	vNetCanInterface.h	vNetCan_getInterface	const vNetCan_interface* (*vNetCan_getInterface_t) (uint32_t version)	Interface version, at the moment 0x010000-00
Ethernet	vNetAutoEthernetInterface.h	vNetAutoEthernet_getInterface	const vNetAutoEthernet_interface* (*vNetAutoEthernet_getInterface_t) (uint32_t version)	Interface version, at the moment 0x010000-00
FkexRay	vNetCanInterface.h	vNetFlexRay_getInterface	const vNetFlexRay_interface* (*vNetFlexRay_getInterface_t) (uint32_t version)	Interface version, at the moment 0x010000-00

Note that the shared library file must be found by the OS API call to load the shared library

- On Windows, by extending the value of the PATH Environment Variable
- On Linux, by extending the value of LD_LIBRARY_PATH (or by putting ESSE libraries in a standard directory)
- On Windows or Linux by providing the Full Path name as the location of the vNet library

5.1.4.1 Example (C code)

Getting Symbols from Shared Library

```
FARPROC getSharedLibrarySymbol(HMODULE module, const char* const sym-
bolName)
{
    return GetProcAddress(module, symbolName);
}

// load the vNetInterface.dll
```

```
HMODULE libhandle = openSharedLibrary();

// get the connection ports of the CAN/CAN-FD networks
vNetCan_getInterface_t pGetInterface = (vNetCan_getInterface_
t) getSharedLibrarySymbol(libhandle, "vNetCan_getInterface");
vNetCan_interface* canVNet = pGetInterface(vNetCanInterfaceVersion);

// get the connection ports of the FlexRay networks
vNetFlexRay_getInterface_t pGetInterface = (vNetFlexRay_getInterface_
t) getSharedLibrarySymbol(libhandle, "vNetFlexRay_getInterface");
vNetFlexRay_interface* flexRayVNet = pGetInterface(vNetFlexRayIn-
terfaceVersion);

// get the connection ports of the Automotive Ethernet networks
vNetAutoEthernet_getInterface_t pGetInterface = (vNetAutoEthernet_
getInterface_t) getSharedLibrarySymbol(libhandle, "vNetAutoEthernet_
getInterface");
vNetAutoEthernet_interface* autoEthVNet = pGetInterface(vNetAutoEth-
ernetInterfaceVersion);
```

5.1.5 VNet Interface Structure

Note that different C structures exist for all three bus types. The following structures exist:

- vNetCan_interface
- vNetAutoEthernet_interface
- vNetFlexRay_interface

As mentioned above there is no common base structure for them. However, all three interfaces objects have the same binary structure comprising

- Member variables such as mVersion
- Member function pointers
- Order of members
- Signatures of the functions the function pointers point to

Note that the approach chosen is a pure C approach: Each vNet interface object is a C struct with C function pointers as members. If you are familiar with C++ you might find this approach a bit similar to C++ classes with a virtual function table.

5.1.6 VNet Ports

vNet allows to open several ports (network interfaces) in each model:

- Each port is an interface to an ETAS Virtual Network
- Each port has a name which is defined as alias name in the XML configuration file. The list of all configured port names in a model can be queried by calling mGetAllPortNames()
- The model has to know which of the ports it wants to open, it must know its own port name in advance (in simple cases it might be sufficient to use the first port name returned by mGetAllPortNames())
- One single model can have many ports, either to the same or to different networks

- If the application process contains many individual models one port can be shared by several models
- Most of the functions in the vNet API are related to ports
  - mOpenPort() is the first function to call, it returns a port handle
  - all subsequent function calls related to that port require its handle as argument
  - mOpenPort() also requires a port configuration which is different for each network type, details re described below
- Each port must be stepped correctly by calling mPreStepPort() and mPostStepPort()
- If the stepping is not correct (wrong rate, wrong points of time) the simulation results will be probably wrong:
  - networks could have different virtual times
  - the simulation can freeze

Getting all vNet port names

The following example shows how to get the name of the own network interface (port) defined in the XML configuration file:

```
// get all Ethernet port names
auto pPortNames = autoEthVNet->mGetAllPortNames();

//get the alias name of the own interface as defined in XML con-
figuration
char* portName = pPortNames[myInterfaceIndex];
```

5.1.7 VNet Port Configuration

Each call to mOpenPort() port requires a port configuration as argument:

- A port configuration is composed of one or more data structures which contain the basic configuration of the network interface
  - Addresses (e. g. MAC addresses, VLAN tags)
  - Flags enabling or disabling certain features of the interface
  - Interface-dependent timing parameters
- Not all of the configuration data need to be initialized
- mOpenPort() is able to fill some configuration variables with initial values from the global configuration file

Example: If the member variable `mStepsPerSecond` is initialized with -1 the function `mOpenPort()` will replace the value -1 by the configured value from the XML configuration file.

- Details about CAN port configuration can be found at "VNET CAN" on page 55
- Details about Automotive Ethernet port configuration can be found at

"vNet Ethernet" on page 62 .

- Details about FlexRay port configuration can be found at "vNet FlexRay" on page 70.

5.1.8 Creating vNet Ports

After setting up a port configuration we can open the port. This task is very similar for all types of networks, so we describe it here. The main differences result from the number and type of configuration structures.

The following table describes which data structures are required to initialize ports using `mOpenPort()`:

Network type	Configuration arguments for **`mOpenPort()`**

Ethernet	`char* portName, vNetAutoEthernetPortConfig* aethCfg,`
	`void* pVendorConfig,`
	`vNetAutoEthernetMacAddress* pMacAddr`

CAN	`char* portName,`
	`vNetCanGlobalConfig* canCfg,`
	`void* pVendorConfig`

FlexRay	`char* portName,`
	`vNetFlexRayGlobalConfig flxGloablCfg*,`
	`vNetFlexRayLocalConfig* flxLocalCfg,`
	`void* pVendorConfig`

> **Note**
>
> Since interface configuration is highly dependent on the network type, the signatures of `mOpenPort()` are different for each network type. This is reasonable because e. g. CAN interfaces do not require any special configuration. In contrast to CAN, FlexRay interfaces need numerous configuration parameters for both, the global network and the local interface being configured. In the case of Ethernet the primary MAC address is required in addition.

5.1.8.1 Opening a vNet port

```
// retrieve a CAN/CAN-FD port handle
vNetCanHandle* pPortHandle = vNetIf->mOpenPort(portName, &canCfg, null-
ptr);

// retrieve a Flexray port handle
vNetFlexRayHandle* pPortHandle = vNetIf->mOpenPort(portName, &flxG-
lobalCfg, &flxLocalCfg, nullptr);

// retrieve the port handle for the Automotive Ethernet port with the
```

```
index myInterfaceIndex
vNetAutoEthernetHandle* pPortHandle = vNetIf->mOpenPort(portName,
aethCfg, nullptr, &aethCfg[myInterfaceIndex]->mSrcMacAdr);

// start the port
pTable->mStartPort(pPortHandle);
```

5.1.9 VNet Interface Callbacks

As mentioned above you will find a lot of function pointers as members in each of the structures describing vNet interfaces.

All of the function pointers will be initialized automatically with pointers to default functions when calling the corresponding `*_getInterface` function.

Some of the function pointers point to functions which allow the user to register own callback functions:

Function pointer name	Signature of the callback	Call of the callback	Comment
`mRegister-RxMessage-Callback`	`rxMessage-Callback_t`	Called for each received frame	This is very important and obligatory in order to receive frames from the vNet interface
`mRegister-RxMessageErrorCallback`	`rxMessageError-Callback_t`	Called for each erroneous frame received by the interface	This allows the user to react on errors
`mRegisterFreeTxBufferCallback`	`freeTxBuffer_t`	Called for each frame after transmission in order to dispose it	This is important when frames need to be freed in a special way. The interface "knows" the point of time at which the frame is not needed any more and calls this callback to dispose it

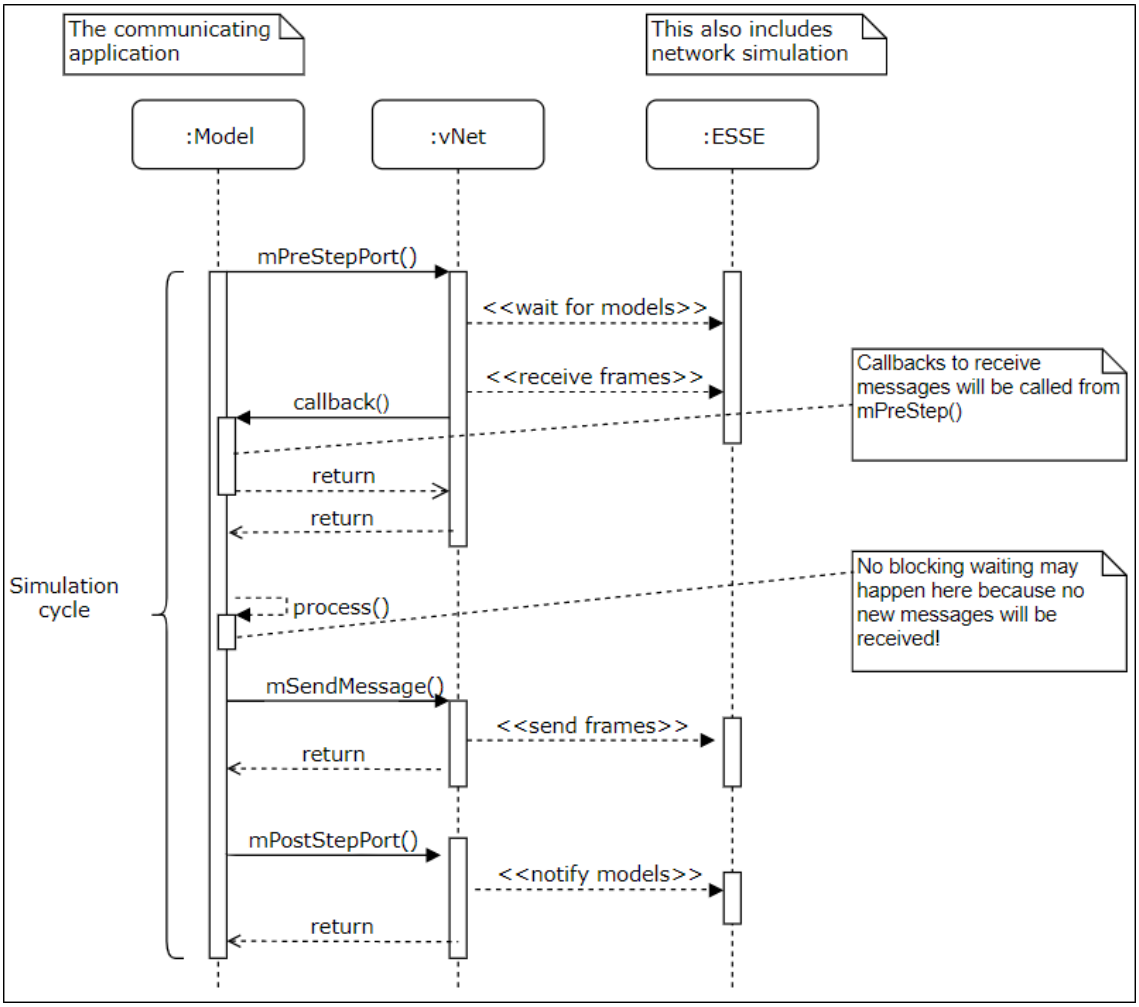
> **Note**
>
> The callbacks will not be called in different thread contexts. ESSE libraries do not start additional threads. All callbacks will be called synchronously in the context of the calling thread during `mPreStepPort()`.

5.1.10 Runtime

- At the beginning of each simulation cycle each interface requires a call of `mPreStepPort()`
  - This call must happen before the actual step() function of the model is called
  - `mPreStepPort()` may block for some time
  - The transmission of frames to destination interfaces in the virtual network happens here
  - All of the communication partners are synchronized by calling `mPreStepPort()`
  - Since frames are received during `mPreStepPort()`, callback function to receive frames might be called here
  - Since errors can happen during transmission the callback function to handle errors might be called in case of errors
  - After `mPreStepPort()` has returned all interfaces might have received new frames. All frames are ready to be received by the application process
  - Between the end of the call to `mPostStep()` and the beginning of the call to `mPreStepPort()` virtual time for the connected network passes - so it jumps to a new value
- The next steps in the model would be to process the received frames
  - Note that after calling `mPreStepPort()` all frames for the current simulation cycle have been received, so no other frames can be expected during this simulation cycle!
    - So waiting for new frames at the application, or intermediate layers interfacing to vNet, layer doesn't really make sense here, it would block until the next network step which cannot occur while the model is executing
    - Avoid blocking waiting of the model for frames - they will never return and the model will get stuck in a deadlock!
  - Frames are ordered in the internal buffer as they have been received. They are arbitrated in time across the previous step and with a comprehensive model of the communication times and prioritized arbitration characteristics for the particular network protocol.
  - The step of the model in which reception occurs is the same for all of messages received (since no further messages can be received until the model step is complete). The frames are already sorted in the order they would have been received (see above) so there is no need to again sort frames received in a step using time stamps. The order in which the receive message callbacks occur should be the order in which the frames are processed.

- After processing all inputs models might send new frames to the network
  - Avoid blocking calls in the interface to vNet (the vNet and SNF layers do not block on sending of frames)
  - Note that sending frames e. g. via sockets might also block if internal queues are full, this behavior must be avoided e. g. by providing special flags to socket functions
  - Note that ETAS Virtual Network simulations know exactly the transmission capacity in each simulation step
  - Note that message loss (for SnfAutoEther and SnfFlexRay networks) may happen if the model attempts to send too many frames in one simulation cycle (if the transmission limit is exceeded). This transmission limit issue for these networks has been adressed for SnfCanMm networks and will be adressed for SnfAutoEther and SnfFlexRay networks in the next development cycle (Q1 2020).
- After sending all frames in the current step the model has to call `mPostStepPort()`
  - No network communication shall happen between `mPostStepPort()` in simulation cycle N and `mPreStepPort()` in simulation cycle N+1!

The sequence chart below demonstrates how a typical simulation cycle with vNet network communication should be implemented:



5.1.11 Message Objects

- Frames are represented by Message Objects. In the vNet API they are represented by different structs and classes for each network type
- Depending on the network type they contain different address information and maximum payload sizes
- Message objects are handled by functions to send and receive frames
- Message objects are represented by C structs or derived C++ classes (when developing in C++ both representations can be used, but the C++ class seems to be more comfortable)
  - Message objects for transmission can be created using the static C++ factory function create(), which allocates the object using the C++ new operator
  - After sending a message its object has to be freed using the C++ delete [] operator
  - Received Message objects, once processed and all references completed, must be freed using the mFreeRxMessage vNet function
- It is also possible to use own functions to allocate/free message objects

(for transmission) without calling the C++ factory function create()

- The following table gives an overview of all message object types:

Network Type	C Struct	Derived C++ class
Automotive Ethernet	`vNetAutoEthernetMessage`	`vNetAutoEthernetMessageEx`
CAN	`vNetCanMessage`	`vNetCanMessageEx`
FlexRay	`vNetFlexRayRxMessage` `vNetFlexRayTxMessage`	`vNetFlexRayRxMessageEx` `vNetFlexRayTxMessageEx`

5.1.12 Sending and Receiving of Frames

- Sending and receiving of frames is similar for all network types
- Differences result mostly from different message objects, payload lengths and address data
- `mSendMessage()` is used to send 1 frame to the network
- It expects
  - a port handle (the port must be open and started)
  - a pointer to a message object holding the address data and the payload, its type depends on the network type
  - Note that some networks such as e. g. Ethernet require additional arguments here
- The callback function registered via `mRegisterRxMessageCallback()` is called for every message received by the interface
  - On frame reception the callback is passed the message object and user callback data
  - User callback data is a void pointer to an object that was registered together with callback by `mRegisterRxMessageCallback()`
  - User callback data is useful to provide user-defined context information to the callback - this is much better than storing this kind of data in static objects

For CAN networks, refer to "Example code to register callbacks and receive frames" section on "VNET CAN" on page 55.

For Automotive Ethernet, refer to "Example code" on "vNet Ethernet" on page 62.

For FlexRay, refer to "Example code" on "vNet FlexRay" on page 70.

5.1.13 Timing

A particular attention must be paid to the timing:

- There is no global simulation time variable
- For each simulation model the most frequently stepped vNet interface shall be used to compute the virtual simulation time
  - by counting the simulation cycles (e. g. between `mPreStepPort()` and `mPostStepPort()`)
  - by multiplying this counter with the duration of each simulation cycle
  - If the most frequently stepped interface is used the simulation time will have the highest accuracy
- The duration of a simulation cycle of each network is configured in the XML global configuration file
- The problem now is: How to provide the cycle duration to the model without reading and parsing the XML file?
- There is a simple solution in vNet:
  - As discussed above each model needs to set-up a port configuration in order to open a port by calling `mOpenPort()`
  - If mStepsPerSecond in the port configuration is set to -1 then `mOpenPort()` will write the configured step duration to mStepsPerSecond
  - This value will then overwrite the initial value -1
  - mStepsPerSecond can then be used to increment the virtual time variable after each step

Typically, the initial virtual time is 0 at the beginning of the simulation, but it can also be set initially to any other value.

All communication partners will count the same virtual time because they synchronize on calling `mPreStepPort()` and `mPostStepPort()`, so all of them shall do exactly the same number of calls if they are connected to the same network. No model can call `mPreStepPort()` more often than any other model connected to the same network.

If a model uses several networks for communication and those networks have different stepping rates, all of the interfaces need to be stepped with the right rates. So at the end all interfaces, no matter what stepping rates they have, must compute the same virtual time!

Do not use any other time sources like system clocks! Do not use any real time clocks!

Example: The following sequence chart shows how the virtual time is kept in all models and network interfaces based on steps. The example has been introduced 5.1.10 on page 49.

In order to understand the sequence chart it is important to mention that in each model and interface the virtual time variable T is local and gets increased at the end of each step (e. g. after calling `postStep()`)

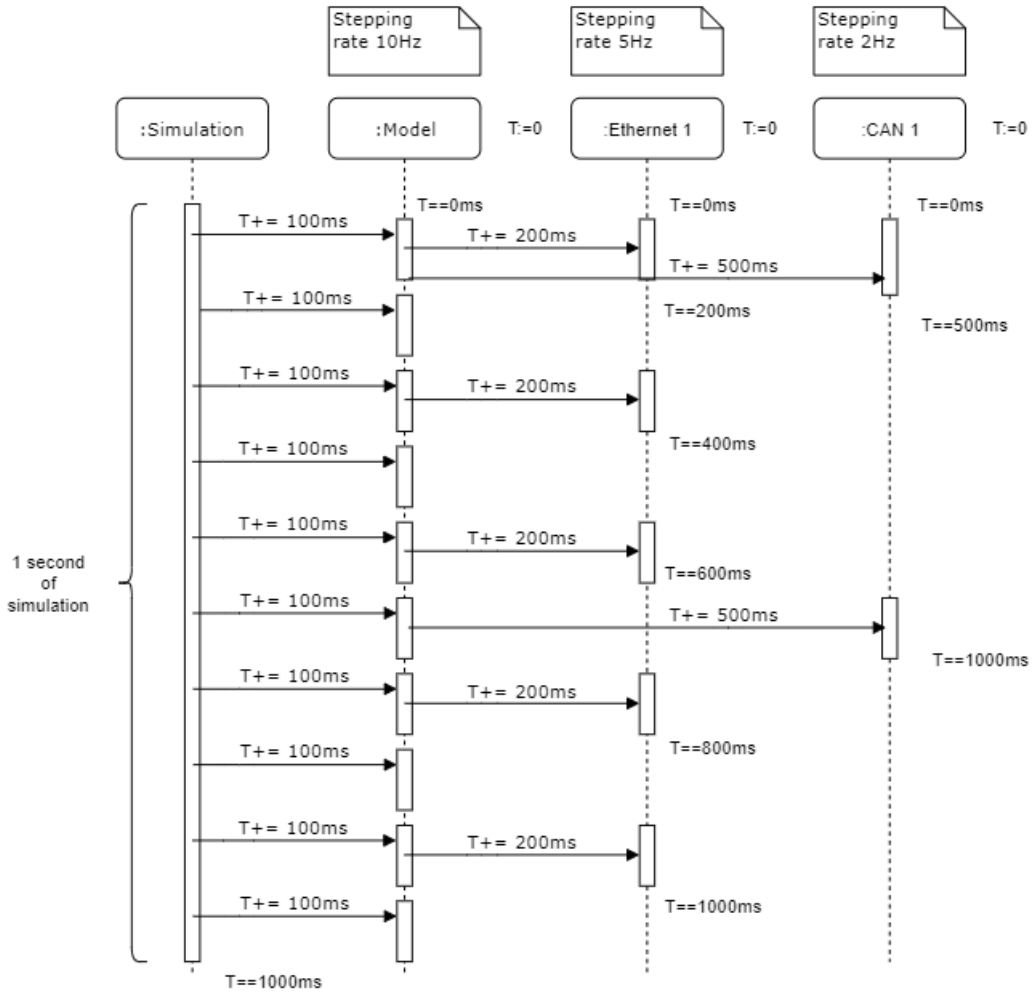


Fig. 5-2: Timing diagram

- At the end of the simulation after 1 second virtual time, the model and both interfaces have measured the same virtual time of 1000ms.
- The models are synchronised only in certain points of time or time intervals
- The virtual time jumps individually for each model and interface according to its step rate
- Models and network interfaces might see different virtual times at the same real point of time because of the jumping and different stepping rates

5.1.14 Clean-up

As mentioned above calls to `mStopPort()` and `mClosePort()` are necessary to stop the simulation. In addition, the handle of the shared library must be closed.

5.2 VNET CAN

In general to apply all vNet interfaces, the facts are explained in "vNet Basics" on page 39. This page aims at outlining the differences between the individual network types and the ESSE implementations of virtual networks. The focus of this page is CAN 2.0 and CAN FD.

The header file used by VNET CAN is `vNetCanInterface.h`.

5.2.1 CAN Facts

- In contrast to Ethernet networks CAN has no concept of addresses and address filters, so **all frames are broadcasted**
- CAN frames are identified by their ID
- Each frame may be sent only by one interface, so for each frame ID there is always at most one interface that is configured to send this ID
- Access to the wire is arbitrated using frame IDs
  - Frames with a lower ID have higher priority and win the arbitration, all sending stations need to do carrier sense and collision detection
  - Frames with a high ID (low priority) might be delayed when high-priority frames are being sent
- Different CAN standards exist
  - classic CAN
    - CAN 2.0A with standard frames with 11 bit frame IDs
    - CAN 2.0B with extended frames with 29 bit frame IDs
  - CAN FD
    - with flexible data rates (frame length may be extended to transport more bytes/frame and thus decrease the relative overhead)
    - the following payload lengths are possible for CAN FD: 0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 32, 48, 64 bytes
    - CAN Fast Data - in this mode after the arbitration is decided the sender and CAN FD receivers switch to a higher baud rate and check for CRC errors, all non CAN FD receivers stop listening
- Many different baud rates can be configured

5.2.2 ESSE Virtual CAN Facts

The ESSE implementation of virtual CAN supports simulations of the logical bus behavior on ISO/OSI layer 2. It doesn't simulate the transmission of voltage levels but the transmission of frames.

The current implementation supports

- Bus arbitration based on frame IDs
- Support for CAN and CAN-FD
- Extended (29 bits) and standard (11 bits) CAN frames, also in a mixed mode
- Configurable baud rates

- Different frame lengths
- Bit stuffing
- Self-reception of frames (internal loop-back)
- Support of various types of reception filters which can be configured independently for each interface

5.2.3 CAN Model

- The basic assumption of the ESSE virtual CAN network is a state-less communication in a single-wire network
- Single-wire means that there are no CAN switches or bridges in the simulated network
- State-less means that the content of the CAN frames does not contain stateful protocols but signal-oriented data
- This assumption is wrong for stateful protocols based on CAN, like e. g. J1939. See the section Limitations below for further explanations,
- CAN arbitration is done according to standard even for mixed, short, and long frame IDs

5.2.4 Initialization

The initialization of the interface is pretty much the same as for vNet Ethernet. The different symbol names to import are listed in the table "Getting Symbols" on page 41 section.

Initialization CAN interface

```
constexpr static const char* dllImportFunc = "vNetCan_getInterface";
#ifdef WIN32
    constexpr static const char* dllName = "vNetInterfaceSnf.dll";
#else //Linux
    constexpr static const char* dllName = "libvNetInterfaceSnf.so";
#endif
    constexpr static int errorBufferSize = 1024;
    constexpr static int ecuIndex = 0;
    char errorBuffer[errorBufferSize];

    //Prepare and setup vNet interface
    //dynamic loading of a shared library
    auto libraryHandle = openSharedLibrary(dllName, errorBufferSize,
errorBuffer);

    //dynamic linking of the factory function
    vNetAutoEthernet_getInterface_t pGetInterface = (vNetCan_getIn-
terface_t) getSharedLibrarySymbol(libraryHandle, dllImportFunc);

    //get the vNet interface object
    auto vNetInterface = pGetInterface(vNetCanInterfaceVersion);
```

5.2.5 Configuration

The configuration of a CAN bus port is simpler than the configuration of an Ethernet or a FlexRay port due to lack of addresses and interface-specific filters.

CAN configuration

```
typedef struct vNetCanGlobalConfig
{
    uint32_t mVersion;
    int32_t mBaudRate;                  // bus clocks per second
    int32_t mFastDataMultiplyer;        // fast bus clocks per bus
clock
    int32_t mStepRate;                  // bus clocks per step
    uint32_t mSpareLocal[4];
} vNetCanGlobalConfig;
```

The main configuration items are:

- *baud rate* of the entire network in bus clocks per second, e. g. 500 000
- *fast data multiplier* - ratio between fast bus ticks and slow bus ticks for the fast data mode in case of CAN FD, e. g. 8
- *step rate* in bus clocks per step, e. g. 1000 so this means a stepping rate of 500 000 / 1000 =500 steps per second (2ms step duration)

The following example demonstrates the configuration of a CAN port:

CAN port configuration

```
// configuration of a CAN/CAN-FD port
vNetCanGlobalConfig canCfg;
canCfg.mBaudRate = 500000;
canCfg.mFastDataMultiplyer = 8;
canCfg.mStepRate = 1000;
```

The port configuration of the CAN/CAN-FD network instance has to be aligned with the related network configuration in the XML file. The baud rate in this example is given by 500k. This matches the configuration ticks=1G and busTicks=2K. The fast data multiplier is equal to the quotient of busTicks=2K and fastBusTicks=250. The parameter `mStepRate` specifies the step rate of the model instance. The given value has to be divided by the baud rate. In the given example this results in a model step width of 2 milliseconds. This matches the network parameters of ticks=1G and stepTicks=2M as discussed.

With such a configuration we can call `mOpenPort()` of the VNET CAN interface to open and start the port as described in the "vNet Basics and example usage with ETAS Virtual Networks" on page 39.

5.2.6 Runtime

The function `mSendMessage()` and the callback function registered via `mRegisterRxMessageCallback()` handle *CAN Message Objects* of type vNetCanMessage:

CAN Message Data

```
typedef struct vNetCanMessage
{
    uint64_t mRequestTicks;             // for Rx message the (Slow)
Bus ticks when message requested
    uint64_t mStartTicks;               // for Rx message the (Slow)
Bus ticks when message started
    uint32_t mMsgLengthTicks;           // for Rx message the (Slow)
Bus ticks it took to transmit message
    uint32_t mCanFrameId;               // 11 or 29 bit Can Frame Id
    uint16_t mCrc;                      // for Rx message the cal-
culated CRC
    vNetCanMessageFlags mFlags;         // vNetCanMessageFlags flags,

    uint8_t  mPayloadLength;            // number of bytes in payload,
for RTR is number of bytes requested.
                                        // MUST be one of 0 to 8, 12,
16, 20, 24, 32, 48 or 64
    uint8_t  mPayload[4];               // expanded to required pay-
load size for frames
} vNetCanMessage;
```

CAN message objects contain the following information in addition to the payload:

- the number of slow bus ticks passed at the point of time when the message was sent to the interface (send request), since this number is an absolute counter value, it might be very large, so a 64 bit number is used here
- the number of slow bus ticks passed at the point of time when the message was arbitrated and sent to the wire, since this number is an absolute counter value, it might be very large, so a 64 bit number is used here
- the number of slow bus ticks it took to transmit the message
- the frame ID as 32bit uint used to transport both, small (11 bit) and large (29 bit) frame IDs
- CRC checksum for received messages only
- Message flags
- payload length in bytes

Note that the length of payload data in mPayLoad is not necessarily 4 bytes, but may be extended to hold larger payloads.

The message flags indicate details about the message to be sent/received:

CAN Message Flags

```
typedef enum evNetCanFlags
{
    eIsExt = 1 << 0,                    // when equal 1 indicates an
extended 29 bit ID instead of a standard 11 bit ID
    eIsRtr = 1 << 1,                    // when equal 1 indicates
Request To Send
    eIsFd  = 1 << 2,                    // when equal 1 indicates FD
mode
    eIsFast= 1 << 3,                    // when equal 1 indicates FD
Fast Data (eIsFd must also be equal 1)
    eIsSelfReception = 1 << 7           // when equal 1 indicates that
the received message was sent from this node
} evNetCanFlags;
```

- `eIsExt` indicates messages with extended header format (29 bit ID)
- `eIsRtr` indicates a request to send (RTR) message
- `eIsFd` indicates the FD mode with flexible data rates
- `eIsFast` indicates the Fast Data mode in which the payload part of the frame was sent with higher baud rate
- `eIsSelfReception` indicates that the message was sent on the same interface (internal loopback), the point of time of reception is when the message was arbitrated (as seen by the virtual model of the particular network protocol)

5.2.7 Sending and Receiving of Messages

Sending of CAN messages is straightforward task for which `mSendMessage()` must be used. Keep in mind that sending is only possible between `mPreStep()` and `mPostStep()`.

The reception of frames and errors, however, requires some preparations. In particular, callback functions must be registered. They will be called in the context of `mPreStep()`.

Receiving of CAN Messages

```
// receiver callback for CAN/CAN-FD messages
int rxMessageCallback(const vNetCanMessage* pMsg, vNetCanUser-
CallbackDataRx* pCallbackData)
{
    // Add code here to process received messages, e. g. to store them
in a buffer

    // free message, it has been processed and is not needed
    pCallbackData->pTable->mFreeRxMessage(pCallbackData->pPortHandle,
pMsg);
    return 0;
}

// error callback for CAN/CAN-FD messages
int rxMessageErrorCallback(const vNetCanMessage* pMsg, const
vNetCanMessageError* errorType, vNetCanUserCallbackDataError* pCall-
```

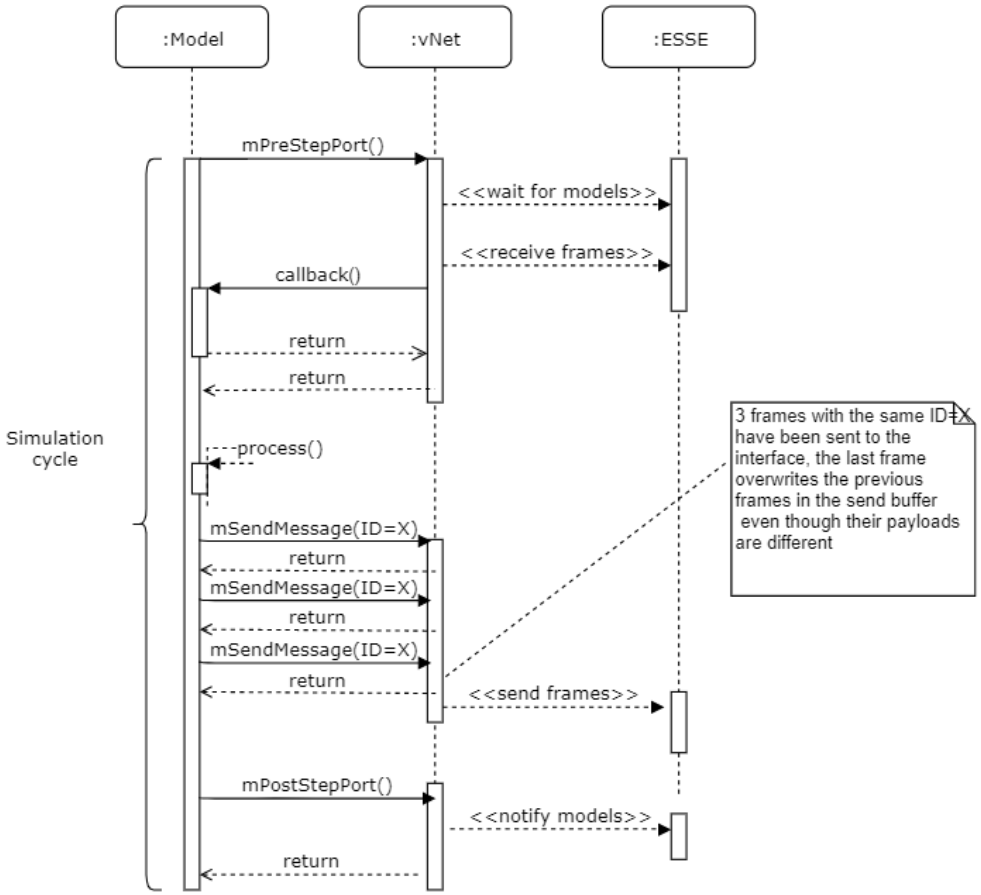
```
backData)
{
    return 0;
}

// free tx buffer callback for CAN/CAN-FD messages
int freeTxBufferCallback(const vNetCanMessage* pTxMsg, vNetCanUser-
CallbackDataTx* pCallbackData)
{
    // DO NOT free - buffer being reused
    return 0;
}
```

Note that there is no sideband structure passed to the callbacks. This is different to Ethernet and FlexRay.

5.2.8 Limitations

As explained above the current CAN implementation is not suitable for stateful communication like e.g . J1939 because frames might overwrite other frames with different content but with the same ID.

- In the case that two frames with the same ID are
  - sent in the same simulation cycle, or, one is sent while one is still pending for transmission from a previous cycle, and,
  - sent from the same CAN interface
- the CAN frame sent later may overwrite the CAN frame sent earlier in the send buffer of the interface


:Model
:vNet
:ESSE
mPreStepPort()
<<wait for models>>
<<receive frames>>
callback()
return
return
Simulation cycle
process()
mSendMessage(ID=X)
return
mSendMessage(ID=X)
return
mSendMessage(ID=X)
return
<<send frames>>
3 frames with the same ID=X have been sent to the interface, the last frame overwrites the previous frames in the send buffer even though their payloads are different
mPostStepPort()
<<notify models>>
return

5.3 vNet Ethernet

In general to apply all vNet interfaces, the facts are explained in "vNet Basics" on page 39. This page aims at outlining the differences between the individual network types and the ESSE implementations of virtual networks.

The header file used by VNET FlexRay: `vNetAutoEthernetInterface.h`.

5.3.1 Ethernet Facts

- In contrast to CAN networks Ethernet uses addresses to direct frames to desired recipients
- Ethernet addresses are also called MAC addresses and are composed of 6 bytes
- A special case are multicast addresses - they allow a frame to be received by all interfaces which share the same multicast MAC address
- Hence, each Ethernet interface may have several MAC addresses assigned to it - the original MAC address, possibly further MAC addresses and several multicast MAC addresses
- A broadcast MAC address is a special case - a frame addressed to a broadcast address is received by all interfaces, the broadcast address doesn't need to be configured, every interface shall know it
- Since every Ethernet interface is permanently listening on the wire it is in principle capable of receiving all frames on the sub-net, not only those addressed to it (promiscuous mode)
- But every Ethernet interface has filters which filter the network traffic and simply ignore Ethernet messages addressed to other hosts
- Automotive Ethernet also requires the support of VLANs
- VLANs is a means to divide an Ethernet network into several virtual sub-networks
- The separation into sub-networks is done by extending Ethernet frame headers by VLAN IDs (and some more data not discussed here)
- Each frame with a VLAN ID X is received only by Ethernet interfaces which have been configured to have the VLAN ID X
- Every Ethernet interface can be assigned many VLAN IDs so each interface may be part of many VLANs
- VLANs also need to be supported by switches

5.3.2 ESSE Virtual Automotive Ethernet Facts

The ESSE implementation of virtual Automotive Ethernet supports simulations of the logical bus behaviour on ISO/OSI layer 2. It doesn't simulate the transmission of voltage levels but the transmission of frames.

It supports:

- Standard Ethernet frames with standard headers
- The common MTU of 1500 bytes
- VLAN header extension with VLAN ID, priority
- Arbitrary number of Ethernet networks with individual stepping rates
- Arbitrary number of Ethernet interfaces connected to each network
- Arbitrary number of MAC addresses configured for each interface
- Arbitrary number of VLAN IDs configured for each interface
- Additional filters can be configured for each interface to decide which frames will be accepted and which will be rejected by the interface
- Self-reception of frames (internal loopback)

5.3.3 Ethernet Model

- The basic assumption of the ETAS Virtual Network is a **single-wire model**
- The single-wire model assumes that all interfaces are connected to one Ethernet segment and thus **compete for access to one wire**
- This assumption leads to a conservative limitation of the number of bytes that can be transmitted per second
- The single-wire model totally ignores the possibility of duplex transmissions, bridges and switches connected to the network
- The model also ignores the possibility of different network latencies due to cable lengths, switches, bridges, switching policies and queuing priorities
- So the assumption is that all frames are transmitted with the same minimum latency, and that network latency is further dependent on the competing network traffic
- According to the information presented in ETAS Virtual Networks the earliest the ESSE Ethernet implementation receives frames into a model is in the next simulation step
- So the minimum latency of frame transmission across models is dependent on the stepping rate - an Ethernet network with a stepping rate of 1000 steps per second (1ms per step) will transmit all frames with a minimum delay of 1ms across models
- Despite the single-wire model there is no CSMA/CD arbitration in the simulated network
- The only arbitration is that a limited number of bits can be transmitted in each simulation step and thus messages that touch this limit are held for transmission in a later network step
- There is also currently a limit to the number of transmission requests a model(s) can make within one simulation step. This limit will be addressed and removed in the next development cycle (Q1 2020).

5.3.4 Initialization Example

The following code snippet shows the creation of an Ethernet interface. This has been discussed on a more abstract level in "vNet Basics" on page 39.

The function openSharedLibrary() mentioned below has two different implementations for Windows and Linux as dynamic loading is highly dependent on the operating system.

AE vNet Code Example

```
constexpr static const char* dllImportFunc = "vNetAutoEthernet_getInterface";
#ifdef WIN32
    constexpr static const char* dllName = "vNetInterfaceSnf.dll";
#else //Linux
    constexpr static const char* dllName = "libvNetInterfaceSnf.so";
#endif

    constexpr static int errorBufferSize = 1024;
    constexpr static int ecuIndex = 0;
    char errorBuffer[errorBufferSize];

    //Prepare and setup vNet interface
    //dynamic loading of a shared library
    auto libraryHandle = openSharedLibrary(dllName, errorBufferSize, errorBuffer);

    //dynamic linking of the factory function
    vNetAutoEthernet_getInterface_t pGetInterface = (vNetAutoEthernet_getInterface_t) getSharedLibrarySymbol(libraryHandle, dllImportFunc);

    //get the vNet interface object
    auto vNetInterface = pGetInterface(vNetAutoEthernetInterfaceVersion);
```

5.3.5 Configuration

As described in "vNet Basics and example usage with ETAS Virtual Networks" on page 39 the configuration of ports is highly dependent on the network type. The configuration of an Ethernet network interface is done by providing a valid port configuration to `mOpenPort()`.

In contrast to other networks Ethernet ports require address information, MAC addresses and VLAN IDs:

- Primary MAC address passed to `mOpenPort()`
- Additional MAC addresses e. g. for MAC multicast
  - either configured in `vNetAutoEthernetPortConfig`
  - or added after `mOpenPort()` via `mAddDestMacAdrFilter()`
- VLAN tags
  - either configured in `vNetAutoEthernetPortConfig`
  - or added after `mOpenPort()` via `mAddVlanIdFilter()`

The structure is named `vNetAutoEthernetPortConfig` and defined in `vNetAutoEthernetInterface.h`. It must contains the following data:

- The maximum baud rate of the ports connection to the vNet or -1
- The source MAC address of the port
- Zero or more VLAN IDs received by this port
- Zero or more Multicast MAC addresses received by this port.

Note that it is also possible to perform some configuration tasks dynamically **after calling** `mOpenPort()` and while the simulation is running:

- `mAddDestMacAdrFilter()` adds an additional multicast MAC address to the interface
- `mAddVlanIdFilter()` adds a VLAN ID to the interface

Both dynamic configuration possibilities are required because **multicast group configuration** is usually dynamic and cannot be anticipated before creating the interface.

So when an interface wants to join an IP multicast group with a *multicast IPv4 address* the corresponding *multicast MAC address* needs to be added to the interface **at runtime**.

As an example a port configuration for 2 is constructed in the following code snipped:

Ethernet Port Configuration

```
// configuration of a list of two Automotive Ethernet ports
vNetAutoEthernetPortConfig* aethCfg[] = {nullptr, nullptr, nullptr };

static int32_t vlanIdsPort1[] = { 2, 3 };

aethCfg[0] = new vNetAutoEthernetPortConfig();
aethCfg[0]->mStepsPerSecond = 1000;
aethCfg[0]->maxSpeed = evNetAutoEthernetAuto100M;
aethCfg[0]->mSrcMacAdr = vNetAutoEthernetMacAddressEx
(0x3e, 0x1a, 0x22, 0x01, 0x23, 0x01);
aethCfg[0]->vlanIdCount = 2;
aethCfg[0]->pVlanId = &vlanIdsPort1[0]; //the pointer points to the
first Id

static int32_t vlanIdPort2 = 4;

aethCfg[1] = new vNetAutoEthernetPortConfig();
aethCfg[1]->mStepsPerSecond = 5000;
aethCfg[1]->maxSpeed = evNetAutoEthernetAuto1G;
aethCfg[1]->mSrcMacAdr = vNetAutoEthernetMacAddressEx
(0x3e, 0x1a, 0x22, 0x01, 0x23, 0x02);
aethCfg[1]->vlanIdCount = 1;
aethCfg[1]->pVlanId = &vlanIdPort2;
```

Remarks

- Note that the array of port configurations must be null-terminated
- So it must have 3 pointers for 2 interfaces, the last one is NULL
- Note that both ports have different speeds and stepping rates because they will be connected to different networks
- Note that as explained before both stepping rates may be initialised to -1 on order to get the stepping rates from the XML configuration
- Note that both ports must have different MAC addresses
- The first port will belong to two VLANs with IDs 2 and 3
- The second port will belong to one VLAN with ID 4

5.3.6 Opening a Port and Additional Configuration

The following example demonstrates how to open a port using configuration data from the previous snipped and how to extend the configuration after opening the port:

Configuration of a vNet Ethernet Interface

```
auto pPortNames = vNetInterface->mGetAllPortNames();
uint8_t addr[] = { 0x11, 0x22, 0x33, 0x44, 0x55, 0x66 };
vNetAutoEthernetMacAddressEx vNetSrcMac( &addr[0] );
auto portHandle = vNetInterface->mOpenPort(pPortNames
[whichPort], &aethCfg[whichPort], nullptr, &vNetSrcMac);
vNetInterface->mStartPort(portHandle);

//as an example add a multicast address to the interface
uint8_t onesMaskData[] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
uint8_t multiCastAddr[] = { 0x01, 0x00, 0x5e, 0xaa, 0xbb, 0xcc };
vNetAutoEthernetMacAddressEx macAddr( &multiCastAddr[0] );
vNetAutoEthernetMacAddressEx macAddrMask( &onesMaskData[0] );
vNetInterface->mAddDestMacAdrFilter
(portHandle, &macAddr, &macAddrMask);

//add the VLAN ID 7 to the interface
uint32_t vLanId = 7;
uint32_t vLanIdMask = 0xffffffff;
vNetInterface->mAddVlanIdFilter(portHandle, vLanId, vLanIdMask );
```

Remarks

- A name must be passed to the port in order to know which interface configuration from the XML configuration file is to be used
- The name must be equal to one of the names returned by `mGetAllPortNames()` which in turn are the alias names from the XML configuration
- The name helps in matching the port to be opened with alias names from the XML configuration file

- By matching the name `mOpenPort()` is able to determine the interface to be opened and the network instance it is connected to
- In the example above one additional multicast MAC address and one additional VLAN ID are added to the interface

5.3.7 Runtime

The following facts have to be kept in mind when operating a Ethernet interface at runtime:

- The function `mSendMessage()` expects a correctly initialized Ethernet message object
- The factory function `vNetAutoEthernetMessage`; Ex: `create()` can create the correct Ethernet header based on given addresses, EtherType and the payload
- After calling the factory function it is possible to manipulate the raw bytes of the returned frame
- The interface doesn't care about the content of the payload
- The payload must not be too large - the MTU limit of 1500 (without header) must be respected
- The payload must not be too small - the minimum frame size of 64 bytes must be respected
- The payload will be padded with additional bytes if it is too short
- Correspondingly, the receiver will receive an Ethernet message object including a fully initialized frame header
- In the sideband structure of type `vNetAutoEthernetSideband` the receiver can find the original payload length as provided by the sender - it does not include possible padding bytes

5.3.8 Sending and Receiving of Messages

Sending of Ethernet messages is straightforward task for which `mSendMessage()` must be used. Keep in mind that sending is only possible between `mPreStep()` and `mPostStep()`.

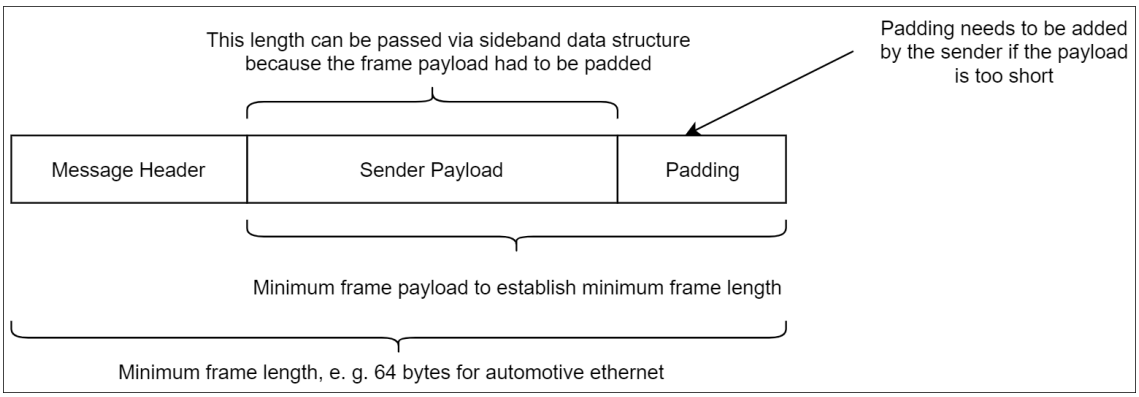
Note that the actual payload length must be passed as argument to the Ethernet version of `mSendMessage()`

- For technical reasons frames must respect minimum lengths, e. g. 64 bytes for Ethernet frames
- If the sender payload is very small the **frame payload must be padded with additional bytes** in order to meet the minimum length requirement.
- Because of this padding the receiver of the frame won't know the exact size of the sender payload
- This problem is solved by providing the RX callback sideband objects as

additional arguments

- Besides of mPayloadLength each sideband object also contains additional useful information

The following figure shows the relationship between frame length, frame paylaod and sender payload:



The reception of frames and errors, requires some preparations. In particular, callback functions must be registered. They will be called in the context of `mPreStep()`.

5.3.9 Receiving of Ethernet messages

```
// receiver callback for Automotive Ethernet messages
int rxMessageCallback(const vNetAutoEthernetMessage* pMsg, const
vNetAutoEthernetSideband* pSideband, vNetEthUserCallbackDataRx* pCall-
backData)
{
    // Add code here to process received messages, e. g. to store them
in a buffer
    // You might want to use pSideband->mPayloadLength to communicate
the original payload length to upper layers

    // free message, it has been processed and is not needed
    pCallbackData->pTable->mFreeRxMessage(pCallbackData->pPortHandle,
pMsg, pSideband);
    return 0;
}

// error callback for Automotive Ethernet messages
int rxMessageErrorCallback(const vNetAutoEthernetMessage* pMsg, const
vNetAutoEthernetSideband* pSideband, const vNetAutoEth-
ernetMessageError* errorType, vNetEthUserCallbackDataError* pCall-
backData)
{
    return 0;
}

// free tx buffer callback for Automotive Ethernet messages
int freeTxBufferCallback(const vNetAutoEthernetMessage* pTxMsg,
vNetEthUserCallbackDataTx* pCallbackData)
{
    // DO NOT free - buffer being reused
    return 0;
}
```

Remarks

- Note that the Ethernet version of the callback has an additional argument: a sideband object
- The sideband object is specific for Ethernet and contains additional data, e. g. sender payload
- User callback provides context information to the callback and is discussed in "vNet Basics and example usage with ETAS Virtual Networks" on page 39.

5.3.10 Future Extensions

- Configurable transmission latencies
- Frame-level access to transmitted and received frames
- Access to frames before and after passing filters

5.4 vNet FlexRay

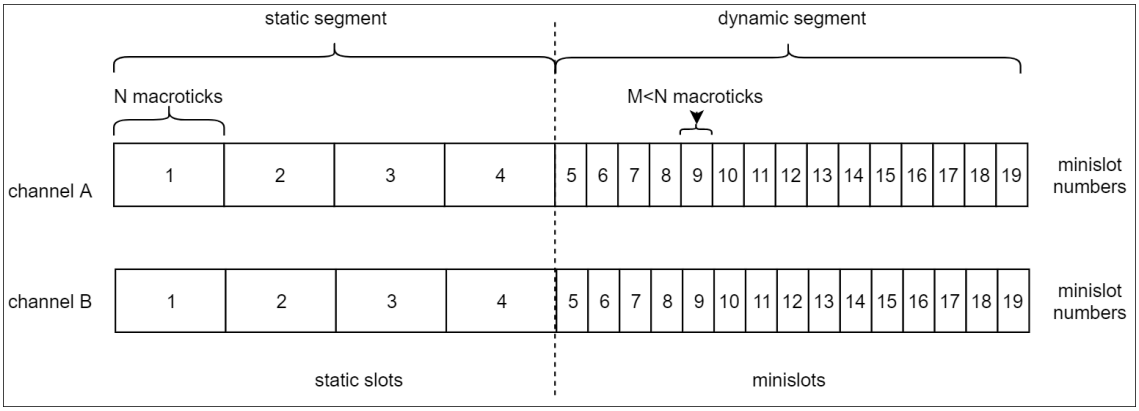
In general to apply all vNet interfaces, the facts are explained in "vNet Basics" on page 39. This page aims at outlining the differences between the individual network types and the ESSE implementations of virtual networks.

The header file used by VNET FlexRay: `vNetFlexRayInterface.h`.

5.4.1 FlexRay Facts

- A FlexRay network requires individual configurations of all stations
- Each station must be configured individually and differently
- These configurations must be done in advance and may not be changed while the network is operated
- The configuration of a FlexRay network requires ca. 30 parameters
- Due to the complexity the configuration is often performed using special configuration tools and stored in a Fibex file
- Manual configuration is difficult and error-prone because it must always be correct: No two messages may be sent at the same time, collisions are not possible and lead to bus errors
- FlexRay communication happens in cycles
- Each cycle has a configurable duration
- The cycle duration is configured as a number of macroticks
- The duration of macroticks is configurable as well
- Each cycle is composed of a static segment, a dynamic segment, a smybol window and network idle time
  - The static segment is composed of N static slots
  - The dynamic segment is composed of M minislots
  - Symbol window and idle time are also parts of a cycle, but not discussed here
- Both sections are used for two types of communication
  - Communication in static (fixed) transmission slots with guaranteed latency (TDMA)
  - Communication in dynamic (variable) time slots without guaranteed latency (FTDMA)
- All slots, static and minislots, have numbers which roughly correspond to CAN frame IDs
- The numbering of slots begins with static slots and continues with minislots
- So in case of N static and M minislots, the slot numbers start with 1 and end with N+M
- Slot numbers < = N are for static slots, slot numbers >N are for minislots

- A message becomes automatically a dynamic message if it is configured for a slot with a number is >N, so it must be a minislot
- Each FlexRay network has 2 channels A and B
- Both channels allow for
  - double transmission capacity or
  - parallel transmission on both channels for the sake of fault-tolerance
- The configuration of macroticks, segments and slots is always the same for both channels A and B
- Messages can be configured to be sent on both channels at the same time



5.4.2 Static Slots

- Each static slot defines an exclusive reserved period of time (in macroticks) in which **a station can send a whole message**
- Static slots are reserved, so a slot remains unused if no station is sending in that slot
- Static slots may suffer from external fragmentation: If only short messages are configured for a static slot and none of them fills it to 100% transmission capacity is wasted
- Each static slots can be assigned several different messages to be sent (cycle multiplexing)
  - In such a case a periodicity is defined for all of the messages assigned to the same slot: a number P which can be 2, 4, 8, 16, 32 or 64
  - Each of the messages sharing the same static slot is assigned an exclusive offset: a number between 0 and P-1
  - So the transmission in such a slot is periodic and repeats every Pth FlexRay cycle
    - If e. g. P=4 then 4 different messages M1, M2, M3, M4 could share the static slot
    - Each of them is sent every in 4th FlexRay cycle
    - Each of them has a different offset: 0, 1, 2, 3
    - The sequence is as follows (cycle #: message) 1:M1, 2:M2, 3:M3, 4:M4, 5:M1, 6:M2, 7:M3, 7:M3 and so on.

5.4.3 Minislots and Dynamic Slots

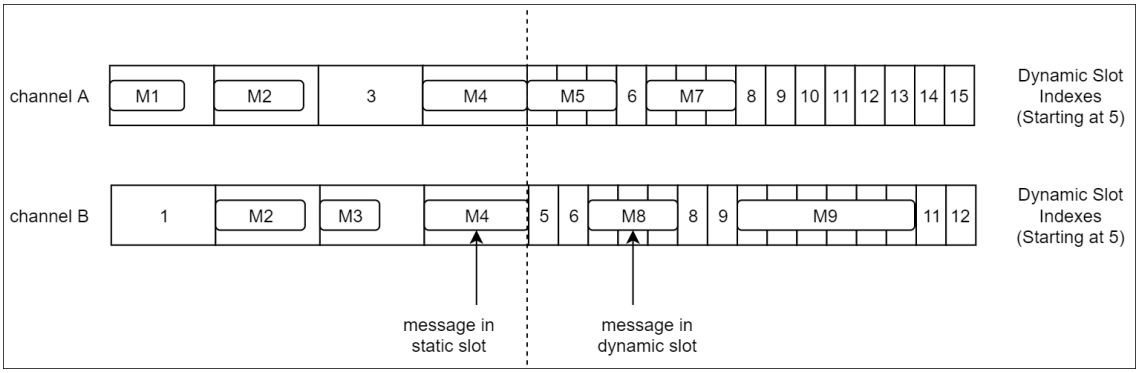
- Messages configured for slot numbers >N are dynamic messages and will be sent in dynamic slots
- Dynamic slots are sequences of minislots in which a dynamic messages is sent
- Minislots define a short reserved period of time (in macroticks) in which a station can **begin to send a message**
- Each station may start sending a message **only in that minislot that was configured for the message**
- Minislots are usually shorter than static slots
- Each dynamic slot is composed of a
  - transmission phase which is followed by a
  - short dynamic slot idle phase
  - both phases consume a sequence of minislots
- The capacity is used more effectively by dynamic slots because only the duration of a minislot is wasted when no message is sent
- Dynamic slots are suitable for event-based communication
- The transmission of a messages in dynamic slot happens in **several subsequent minislots without preemption**
- That means: When a station begins to send a message in a minislot, it might consume **as many subsequent minislots as needed to transmit the complete message**
- So when the transmission has started in a minislot, all other stations must wait until it has been completed. After transmission in a dynamic slot the station with the next dynamic slot index may start sending. The dynamic slot index does not increment during additional minislots needed to complete the previously transmitted message.
- Each station which wants to start sending in a minislot must check whether the number of minislots left in the current cycle is sufficient to transmit its message
- If not, the transmission of the message must be postponed to next cycle, the dynamic slot index is incremented, and the remaining minislots may be sufficient to accommodate a message with a matching Dynamic Slot Index
- Due to these facts dynamic messages have priorities: messages configured for Dynamic Slot Indexes with lower number have a higher priority than messages configured for Dynamic Slot Indexes with high numbers

5.4.4 Schedule example

The following example shows a schedule containing messages M1,..., M9 configured for both, static and dynamic segments and for both channels.

For instance, M2 and M4 are sent simultaneously on both channels.

We assume that M6 has lost its dynamic slot and cannot be sent in this cycle. This phenomenon will be explained in the next sub-section.



Note

- The waste of static slot capacity in slots 1, 2, 3: scheduled messages M1, M2, M3 do not fill the static slots.
- The waste of minislots.
  - when no message was configured for a minislot.
  - when a configured message could not be sent because there are insufficient minislots remaining in the cycle.

5.4.5 Priority in Dynamic Slots

Note the method how dynamic slots are allocated. The following example, (for the dynamic slot transmissions shown above, shall demonstrate this:

After sending all static messages in static slots, the transmission of the dynamic segment (15 minislots) begins.

The following dynamic messages are scheduled to be transmitted in the dynamic segment of a FlexRay cycle:

Message Name	Dynamic Slot Index	Length in Mini Slots	Channel for Transmission
M5	5	3	A
M6	6	14	A
M7	7	3	A
M8	7	3	B
M9	10	6	B
M10	11	3	B

The dynamically allocated minislots for channel A would be allocated as shown:

Mini Slot Number	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Dynamic Slot Index	5	5	5	6	7	7	7	8	9	10	11	12	13	14	15
Message Name	M5	M5	M5	--	M7	M7	M7	--	--	--	--	--	--	--	--

M5 was transmitted in dynamic slot index 5 taking up 3 minislots for its transmission in minislots 5, 6, 7.

Because of this the transmission of M6 could not begin in its Dyamic Slot Index 6 (minislot 8) as there were only 12 minislots remaining in the cycle, and no message was transmitted in minislot 8.

Since M6 could not be sent, the transmission of M7 could start in minislot 9.

The priorities of M6 and M7 are in this case inverted: Although M6 has actually a higher priority (lower dynamic slot 6), M7 was sent in the current cycle instead of M6. In a subsequent cycle, if M5 was not requested for transmission, then M6 could start in minislot 6 which would then be Dynamic slot 6 and it would consume the remaining minislots for the cycle.

The dynamically allocated minislots for channel B would be allocated as shown:

Mini Slot Number	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Dynamic Slot Index	5	6	7	7	7	8	9	10	10	10	10	10	10	11	12
Message Name	--	--	M8	M8	M8	--	--	M9	M9	M9	M9	M9	M9	--	--

M8 was transmitted in dynamic slot index 7 taking up 3 minislots for its transmission in minislots 7, 8, 9.

M9 was transmitted in dynamic slot index 10 taking up 6 minislots for its transmission in minislots 12, 13, 14, 15, 16, 17.

Because of the transmissions of M8 and M9, M10 could not begin in its transmission in Dyamic Slot Index 11 (minislot 18) as there were only 2 minislots remaining in the cycle, and no message was transmitted in minislots 18 and 19.

5.4.6 ESSE Virtual FlexRay Facts

- The ESSE virtual FlexRay implementation supports all of the configuration features mentioned above.
- Particular attention must be paid to the stepping - see the section Stepping below.
- Due to the complex configuration described below the user of configuration tools is advised.

5.4.7 ESSE FlexRay Model

ESS FlexRay virtual networks have a simple single-wire line topology.

Initialization

The initialization of the interface is pretty much the same as for vNet Ethernet and VNET CAN. The different symbol names to import are listed in the table "Getting Symbols" on page 41.

Configuration

The configuration of a FlexRay port is quite complex and requires a lot of parameters. The best way to have a correct configuration is to use a FlexRay configuration tool.

Once the data is available, a global configuration for the entire network and and one local configuration for each vNet port must be created.

FlexRay config

```
typedef struct vNetFlexRayGlobalConfig
{
    // header
    uint32_t mVersion;
    int32_t mBusClocksPerSec;           // Baud rate
    int32_t mBusClocksPerMacroTick;     // Macro tick rate
    int32_t mMacroTicksPerStep;         // macroTicks per Step

    // cycle
    uint32_t mgMacroPerCycle;           // macroTicks per cycle
    uint32_t mgSpareCycle[3];

    // static segment
    uint32_t mgNumberOfStaticSlots;     // Number of static slots in
static segment
    uint32_t mgdStaticSlot;             // Number of macroTicks in
each each static slot
    uint32_t mgPayloadLengthStatic;     // Number of 2 byte words of
payload in each static slot
    uint32_t mgSpareStatic[1];

    // dynamic segment
    uint32_t mgNumberOfMinislots;       // Number of mini slots in
dynamic segment
    uint32_t mgdMinislot;               // Number of macroTicks in
each mini slot
```

```
    uint32_t mgdDynamicSlotIdlePhase;   // Number of macroTicks of
idle after a dynamic slot
    uint32_t mgSpareDynamic[1];

    // symbol segment
    uint32_t mgdSymbolWindow;           // Number of macroTicks in sym-
bol segment
    uint32_t mgSpareSymbol[3];

    uint32_t reserved[12];
} vNetFlexRayGlobalConfig;
```

FlexRay Main configuration

The main configuration items are:

- **baud rate** of the entire network in bus clocks per second, e. g. 10 000 000
- **bus clocks per macrotick** - the duration of a macrotick in bus clock ticks
- **macroticks per step** - the number of macroticks per simulation step, together with the number of bus ticks per macrotick we can compute the overall step duration and the stepping rate
- **macroticks per cycle** - defines the cycle duration in macroticks
- **number of static slots** - how many static slots are in a static segment
- **payload length static** - number of 2 byte words of payload in each static slot, **the computation of this value is tricky** because it is coupled with the length and number of macroticks in a static slot
- **number of minislots** - the number of minislots in the dynamic segment
- **minislot** - number of macroticks per minislot
- **dynamic slot idle phase** - the length of the idle phase of a dynamic slot in macroticks
- **symbol window** - number of macroticks in the symbol segment

FlexRay local configuration

The local configuration is shown below:

```
typedef struct vNetFlexRayLocalConfig
{
    // local config
    uint32_t mpLatestTx;                // the last mini slot to start
a dynamic slot
    uint32_t mpSpareLocal[3];
} vNetFlexRayLocalConfig;
```

The only interesting configuration item here is latestTx. It contains the number of the latest minislot in which the transmission of a dynamic message may start. Such a value should be computed individually for each interface depending of the lengths of messages to be sent.

Creating Flexray Configuration

The creation of both configuration structures is straightforward provided that all data is available:

```
vNetFlexRayGlobalConfig flxGlobalConfig;
flxGlobalConfig.mBusClocksPerSec = 10000000;
flxGlobalConfig.mBusClocksPerMacroTick = 1;
flxGlobalConfig.mMacroTicksPerStep = 5000;
flxGlobalConfig.mgMacroPerCycle = 5000;
flxGlobalConfig.mgNumberOfStaticSlots = 91;
flxGlobalConfig.mgdStaticSlot = 24;
flxGlobalConfig.mgPayloadLengthStatic = 8;
flxGlobalConfig.mgNumberOfMinislots = 289;
flxGlobalConfig.mgdMinislot = 5;
flxGlobalConfig.mgdDynamicSlotIdlePhase = 1;
flxGlobalConfig.mgdSymbolWindow = 0;

vNetFlexRayLocalConfig flxLocalConfig;
flxLocalConfig.mpLatestTx = 249;
```

The configuration example is described below:

- The FlexRay network is a 10 MBit network because of 10000000 bus clocks per second
- Each macrotick takes 1 bus clock = 100 ns
- Each simulation step takes 5000 macrotocks = 500 us = 0.5 ms so we 2000 simulation steps per second
- Each FlexRay cycle requires also 5000 macroticks
- So in the special case the length of a simulation cycle is equal to the length of a bus cycle. This is makes the understanding of the simulation easy, but is actually not required (see discussion about simulation cycles below)
- Each FlexRay cycle has 91 static slots, each of which takes 24 macroticks
- So the static segment comprises 91 * 24 = 2184 macroticks
- The payload length in each static slot is 8 * 2 = 16 bytes
- The number of minislots is 289, each of them takes 5 macroticks
- So the dynamic segment is 289 * 5 = 1445 macroticks
- The symbol window stakes 249 macroticks
- Plausibility check: 2184 + 1445 + 249 = 3878 macroticks < 5000 macroticks configured for each FlexRay cycle

5.4.8 Opening FlexRay Ports

// Choose e. g. the first interface from the list to open

```
// Choose e. g. the first interface from the list to open
auto pPortNames = flxRayIface->mGetAllPortNames();
char* portName = pPortNames[0];
```

```
// retrieve a Flexray port handle
vNetFlexRayHandle* pPortHandle = flxRayIface->mOpenPort(
portName, &flxGlobalCfg, &flxLocalCfg, nullptr);
```

5.4.9 Sending and Receiving of Messages

- The general rule that messages transmitted in simulation step N can, at the earliest, only be received in step N+1 applies also to FlexRay.
- Note that regarding the communication mechanism in FlexRay we need to consider 3 points of time when we talk about simulated message transmission:
  - The time when the message was sent to the FlexRay network interface - it is the **send request time**
  - The time when the message was transmitted on the wire in the configured time slot - it is the **transmission time**
  - The time when the message is received on the remote network interface - it is the **reception time**
- Note that several FlexRay cycles may pass between the send request and the actual transmission - time interval between 1 and 2
- Also note that the time interval between 2 and 3 is always 1 simulation step (as explained above)
- The reason for this is explained in the next sub-section
- Note also that applications do not need to respect segments and slot configurations: **Applications can send any messages any time to a FlexRay network interface**
- **It is the responsibility of the vNet network interface to store the send request and to transmit the messages as early as possible in the next possible slot**

5.4.10 FlexRay Cycles and Simulation Cycles

Facts and Findings

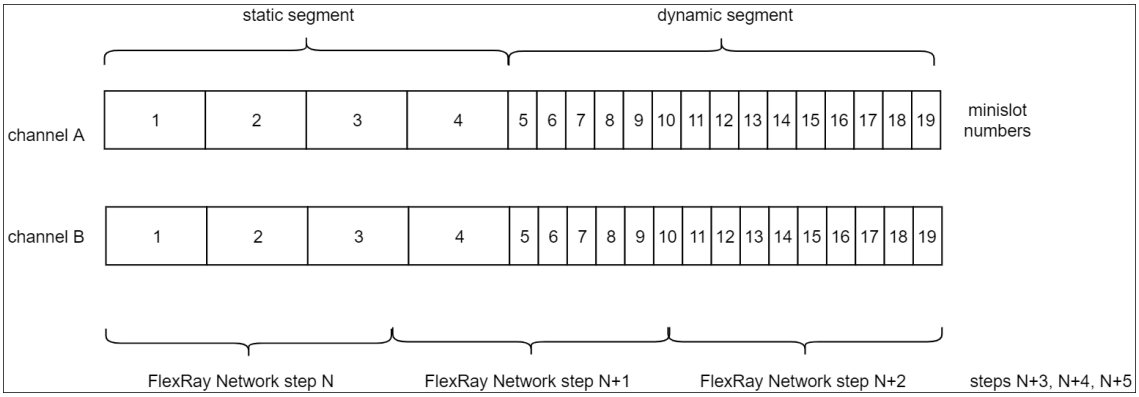
- FlexRay is different with respect to the fact that messages are scheduled in certain slots during FlexRay cycles.
- The global configuration data defines the length of a simulation cycle as a multiple of macroticks. **This means that simulation steps may never start or end in the middle of a macrotick**.
- However, it is possible for a FlexRay cycle to comprise several simulation steps. See the example below
- It is not allowed to define "large" simulation step sizes which would comprise several FlexRay cycles. This would lead to incorrect simulation results.

- It is allowed for simulation steps to start and end in the middle of
  - a static segment,
  - a dynamic segment,
  - a static slot and
  - a dynamic slot.

Example Schedule

The figure below shows a FlexRay cycle which is covered by 3 simulation cycles.

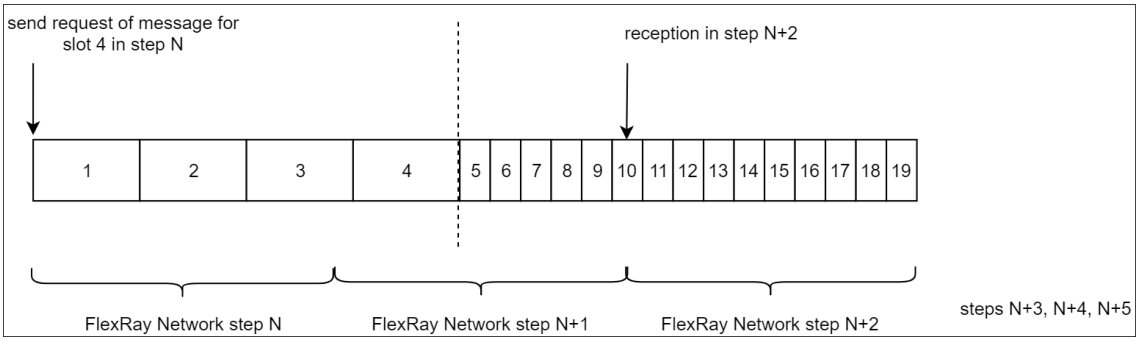
- Simulation step N covers static slots 1, 2 and a part of 3
- Simulation step N+1 covers a part of slot 3, slot 4, minislots 5-9 and a part of minislot 9
- Simulation step N+2 covers a part of minislot 10 and minislots 11-19
- The same applies to the following simulation steps N+3, N+4 and N+5 - this schema repeats every 3rd FlexRay cycle



This flexibility has a little bit surprising implications when sending and receiving messages:

- A message configured for static slot 1 and sent in step N will be received in step N+1
- A message configured for static slot 4 and sent in step N will be received in step N+2. This example is shown in the next figure below.
- A message configured for static slot 3 and sent in step N will be received in step N+2 because slot 3 is not entirely covered by step N
- A message configured for static slot 3 and sent in step N+1 will be received in step N+5 because slot 3 is not entirely covered by step N and the message has already missed its static slot in N+1
- Similar restrictions apply for the reception of dynamic messages, however, this is further complicated by the aggregation of minislots into dynamic slots which depends on transmission requests for messages with higher priority dynamic indexes

The following figure depicts how the three previously mentioned points of time are related to each other:


send request of message for slot 4 in step N
reception in step N+2
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
FlexRay Network step N
FlexRay Network step N+1
FlexRay Network step N+2
steps N+3, N+4, N+5

5.5 VNET LIN

5.5.1 LIN Bus Basics

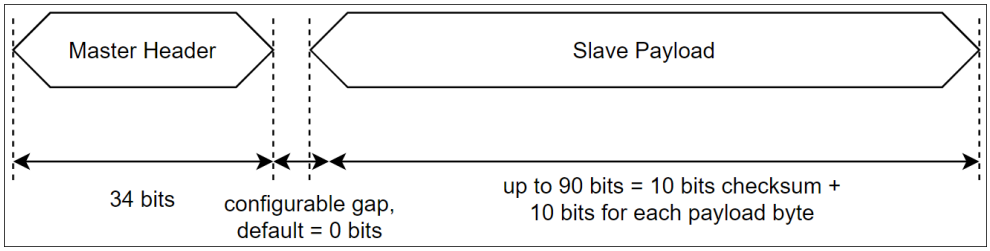
- There is always 1 Master node and possibly many Slave nodes on a LIN bus
- The Master node can also be a Slave
- The Master node has a schedule, when it is running it follows that schedule and sends requests to Slaves
  - In that schedule, the Master requests information from Slaves
  - It sends a Master request with a LIN message ID
  - If any of the Slaves is configured to respond to such an ID, it must respond immediately after the Master request
  - A Slave can respond to many IDs as requested by the Master
  - After a Slave has responded to Master, the Master can send a new request to the same or another Slave
  - If no Slave responds the LIN bus, it is idle until the Master sends a new request

5.5.2 LIN Frames

LIN messages have a simple structure and a very limited payload of maximum 8 bytes.

The Master header and the Slave payload are two messages which form one single frame!

So there is 1 frame composed of two messages coming from two different LIN end points.



According to the diagram above, the maximum frame length (without the additional gap in between) is 34 bits + 90 bits = 124 bits.

5.5.3 Baud Rates

Typical baud rates for LIN buses are:

- 19200 baud
- 9600 baud
- 4800 baud
- 2400 baud
- 1200 baud

The LIN bus implementation also supports other baud rates. There is no limitation.

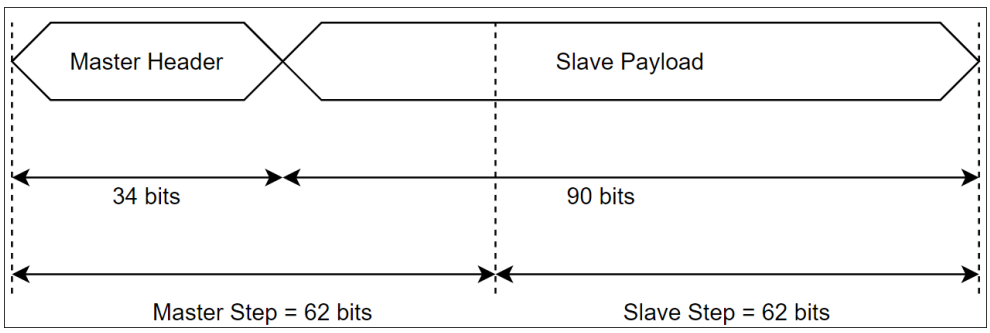
5.5.4 LIN Frame Transmission

The points below describes how the frame transmission in ESSE/vNet libraries.

- The Master header and Slave payload are always sent in 2 subsequent simulation steps
- The Slave payload is always scheduled immediately after the Master header.
- The small gap between both parts is by default 0 bits (can be configured to be a higher value)
- This means that we must always consider two subsequent steps to transmit both, Master header and Slave payload. This is called as a Master-Slave pair.
- If there is no Slave that could respond to the Master request, it is possible that the Slave message is empty (no response)

5.5.5 Perfect LIN Configuration

- In a perfect LIN configuration of 8 bytes of payload, two subsequent simulation steps, i.e., 1 Master step and 1 Slave step - must transport exactly 124 bits
  - That means that each simulation step must transport exactly 62 bits.
  - A perfect step rate of 8 bytes of payload is defined such that it transports exactly 62 bits in each step.
- In a simple configuration of 4 bytes of payload, two subsequent simulation steps must transport exactly 84 bits
  - That means that each simulation step must comprises exactly 42 bits.
  - A perfect step rate of 4 bytes of payload is defined such that it transports exactly 42 bits in each step.
- In such perfect configurations with perfect step rates, there is no waste of bits. This is because each Master-Slave pair of steps comprises exactly the number of bits it needs.

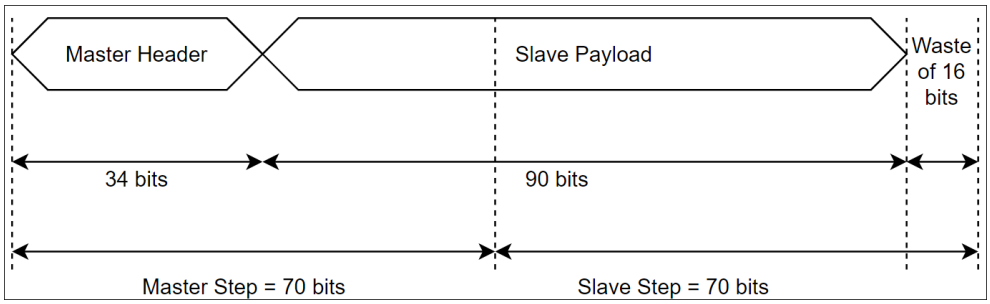


5.5.6 Simple LIN Configuration

- In a simple LIN configuration, it transmits the perfect number of bits or more bits in each simulation step. Hence, for 8 bytes of payload, must transmit at least 62 bits in each step
- Every additional bit on top of those 62 bits is wasted, but possible.
- This means that step-rates are lower than the perfect step rate which will lead to waste of bits in each Master-Slave pair
- This waste is accepted in typical configurations because Master transmission schedules have typically a rough raster of 5ms or 10ms
- The benefit of such simple configurations is that they are easy to understand and the send behavior is easy to track

In the following diagram, it is chosen a slightly lower step rate which means that each step comprises of 70 bits instead of 62 bits.

With these 70 bits, waste of 16 bits after each Master-Slave pair.



5.5.7 Baud Rates and Step Rates

Bits Transmitted in Each Step - 62 bits would be perfect for 8 bytes					Perfect Step Rates for payload bytes (steps/s)		
Baud rate	200 (steps/-s)	100 (steps/-s)	50 (steps/-s)	25 (steps/-s)	8 bytes (62 bit-s/step)	4 bytes (42 bit-s/step)	2 bytes (32 bit-s/step)
2000-0	100	200	400	800	322	476	625
19200	96	192	384	768	309	457	600
10000	50	100	200	400	161	238	312
9600	48	96	192	384	154	228	300
4800	24	48	96	192	77	114	150
2400	12	24	48	96	38	57	75
1200	6	12	24	48	19	28	37

Tab. 5-1: Baud Rates and Step Rates

The colors specified in the table above are defined as follows:

- Dark Green - Simple Step Rates for 8 bytes payload (each step has at least 62 bits)
- Light Green - Simple Step rates for 2 and 4 bytes of payload (each step has at least 32 bits)
- Blue - Perfect Step Rates (each step has approx. 62 bits)
- Yellow - Not perfect step rates - Master-Slave Pairs need more than 2 simulation steps

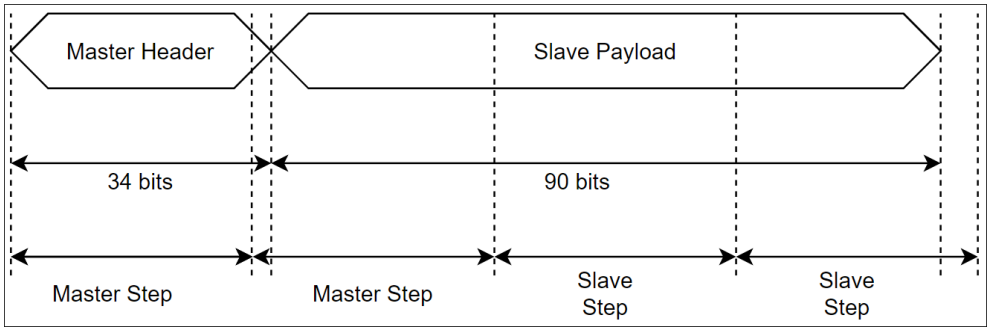
5.5.8 Not Perfect Step Rates

Can we choose high step rates which lead to yellow bit numbers in the table above?

- Yes they can be safely chosen, the simulation will work
- But, Master-Slave pairs might need more than 2 subsequent steps for a complete transmission. Such kind of settings are safe and are supported
- The only downside is that such non-perfect settings are a bit harder to understand and to debug

In the following diagram, the step rate has been doubled. Each Master message requires 2 steps now (a remainder of the Master message will be transmitted in step 2).

Correspondingly, the Slave message also requires 2 simulation step. After finishing the transmission of the Slave part, a few bits are wasted.



How to use the Tab. 5-1?

Questions	Answers
Example #1	
To design a LIN bus with 19200 bits/s. The maximum payload to transport is 4 bytes.	

What is the Perfect Step Rate?	In line with baud rate 19200 which you can find in the second blue column the perfect step rate of 457 steps/second.
How to choose a Simple Step Rate?	A good simple step rate would be 400 steps/sec = 2.5 ms for each simulation step. Any lower step rate would also be okay.
Example #2	
To design a LIN bus with 9600 bits/s and a step rate of 100 steps/s (10ms each step). The maximum expected payload is 8 bytes.	
Have we chosen a Perfect Step Rate?	No, the perfect step rate would be 154 steps/s. This is more than 100 steps/s we have chosen.
Do we have a simple step rate?	Yes, according to the second green column, it has a capacity of 96 bits in each step. This is more than the perfect bit number of 62 bits/step for 8 bytes of payload.
How many bits can we transmit in each step?	In each step, it can transmit 96 bits, 192 bits in total in each Master-Slave Pair
How many bits do we waste in each step?	192 bits - 124 bits = 68 bits are wasted in each Master-Slave Pair, that is 34 bits waste in each simulation step.

Tab. 5-2: Tab. 5-1 usage

6 Contact Information

Technical Support

For details of your local sales office as well as your local technical support team and product hotlines, take a look at the ETAS website:

www.etas.com/hotlines

ETAS Headquarters

ETAS GmbH

Borsigstraße 24	Phone:	+49 711 3423-0
70469 Stuttgart	Fax:	+49 711 3423-2106
Germany	Internet:	www.etas.com

Glossary

A

API

Application Programming Interface

C

COSYM

Co-simulation of Systems. COSYM is built to integrate and simulate mechatronic systems consisting of control as well as plant models.

D

DSE

Distributed Simulation Engine

DSM

Distributed Simulation Manager

F

FMU

Functional Mockup Unit. FMUs compliant with version 2.0 of the FMI standard for co-simulation.

S

SNF networks

System Network Fabrics

SnfSyncSignals

Network Fabric SyncSignals

V

vNET

Virtual Network