

ETAS VECU-BUILDER V1.6



User Guide

Copyright

The data in this document may not be altered or amended without special notification from ETAS GmbH. ETAS GmbH undertakes no further obligation in relation to this document. The software described in it can only be used if the customer is in possession of a general license agreement or single license. Using and copying is only allowed in concurrence with the specifications stipulated in the contract.

Under no circumstances may any part of this document be copied, reproduced, transmitted, stored in a retrieval system or translated into another language without the express written permission of ETAS GmbH.

© Copyright 2024 ETAS GmbH, Stuttgart

The names and designations used in this document are trademarks or brands belonging to the respective owners.

VECU-BUILDER V1.6 | User Guide R11 EN | 04.2024

Contents

| | | |
|-------|---|----|
| 1 | Introduction | 6 |
| 1.1 | Intended Use | 6 |
| 1.2 | Target Group | 6 |
| 1.3 | Data Protection | 6 |
| 1.4 | Data and Information Security | 6 |
| 1.4.1 | Data and Storage Locations | 6 |
| 1.4.2 | Technical and Organizational Measures | 7 |
| 2 | About VECU-BUILDER | 8 |
| 2.1 | VECUC-BUILDER on YouTube | 8 |
| 2.2 | Warning and Error Messages | 9 |
| 2.3 | Basics | 9 |
| 2.4 | Virtual ECU | 10 |
| 2.5 | vECU Creation Process Workflow | 10 |
| 2.6 | Functional Mock-up Interface (FMI) | 11 |
| 3 | Installation | 12 |
| 3.1 | Hardware Requirements | 12 |
| 3.2 | Preparation | 12 |
| 3.3 | Installation Content | 12 |
| 3.4 | Licensing | 13 |
| 3.5 | Installation on Windows 10 | 13 |
| 3.5.1 | Software Requirements for Windows 10 | 13 |
| 3.5.2 | Manual Installation of VECU-BUILDER | 14 |
| 3.5.3 | Silent Installation of VECU-BUILDER | 15 |
| 3.5.4 | Uninstalling VECU-BUILDER on Windows 10 | 16 |
| 3.6 | Installation on Ubuntu 20.04 LTS | 17 |
| 3.6.1 | Software Requirements for Ubuntu 20.04 LTS | 17 |
| 3.6.2 | Installing License Manager (LiMa) on Ubuntu 20.04 LTS | 17 |
| 3.6.3 | Opening ETAS License Manager on Ubuntu 20.04 LTS | 18 |
| 3.6.4 | Activating the LiMa License | 18 |
| 3.6.5 | Installing VECU-BUILDER on Ubuntu 20.04 LTS | 19 |
| 3.6.6 | Uninstalling VECU-BUILDER on Ubuntu 20.04 LTS | 19 |
| 3.7 | Installation on Ubuntu 20.04 for WSL | 20 |
| 3.7.1 | Software Requirements for Ubuntu 20.04 on WSL | 20 |
| 3.7.2 | Installing WSL on Windows | 20 |

| | | |
|---------|--|----|
| 3.7.3 | Installing Ubuntu 20.04 on WSL | 20 |
| 3.7.4 | Installing Dependent Software Packages | 21 |
| 3.7.5 | Installing License Manager (LiMa) on Ubuntu 20.04 for WSL | 22 |
| 3.7.6 | Installing VECU-BUILDER on Ubuntu 20.04 for WSL | 22 |
| 3.7.6.1 | Opening ETAS License Manager on Ubuntu 20.04 LTS for WSL | 23 |
| 3.7.7 | Activating the LiMa License | 24 |
| 3.7.8 | Uninstalling VECU-BUILDER on Ubuntu 20.04 for WSL | 24 |
| 3.8 | Installed Files and Folders | 25 |
| 4 | Working with VECU-BUILDER | 28 |
| 4.1 | Creating a New Workspace | 29 |
| 4.1.1 | Creating a Workspace on Windows | 29 |
| 4.1.2 | Creating a Workspace on Ubuntu 20.04 LTS | 31 |
| 4.2 | Importing Files and Folders | 33 |
| 4.3 | Building the vECU | 35 |
| 4.4 | Building the FMU | 36 |
| 4.5 | Workspace Content | 37 |
| 4.6 | Configuration | 39 |
| 5 | Exploring the Examples/Templates | 50 |
| 5.1 | Simple Example | 50 |
| 5.1.1 | FMPy | 50 |
| 5.1.2 | Difference Between Debug and Release vECUs | 52 |
| 5.1.2.1 | Keeping Symbol Information in a Release FMU | 53 |
| 5.1.3 | InitialData Functionality | 53 |
| 5.1.4 | EEPROM Functionality | 57 |
| 5.1.5 | Features to Explore in the Simple Example Workspace | 59 |
| 5.2 | BCU Example (Only Available for Windows) | 60 |
| 5.2.1 | Show Symbol Information | 61 |
| 5.2.2 | A2L File Patching | 61 |
| 5.2.3 | A2L Name Mapping | 62 |
| 5.2.4 | Hex File Generation | 63 |
| 5.2.5 | Example of Additional Scripts: A2L Characteristics as Parameters | 64 |
| 5.2.6 | Features to Explore in the BCU Workspace | 65 |
| 5.3 | EventTrigger Example | 66 |
| 5.3.1 | Event-Triggered Tasks | 66 |
| 5.4 | Template For Plugin V1 (FMI2) | 68 |
| 5.4.1 | Plugin Feature | 68 |
| 5.4.2 | Plugin Configuration | 69 |

| | | |
|-------|--|-----|
| 5.4.3 | Plugin Interface | 70 |
| 5.4.4 | What a Plugin Can Do With Tasks | 72 |
| 5.5 | Template For Plugin V2 (FMI3) | 73 |
| 5.5.1 | Plugin Feature V2 | 73 |
| 5.5.2 | Plugin Configuration V2 | 74 |
| 5.5.3 | Plugin Interface V2 | 75 |
| 5.5.4 | What a Plugin Can Do With Tasks V2 | 77 |
| 6 | Controlling VECU-BUILDER | 78 |
| 6.1 | Manual Interaction | 78 |
| 6.2 | Command-Line Interface | 78 |
| 6.3 | Ubuntu 20.04 on WSL Command Line Interface | 81 |
| 7 | Debugging vECU | 82 |
| 7.1 | Debugging with Visual Studio 2019 | 83 |
| 7.2 | Debugging with Visual Studio Code | 84 |
| 8 | Troubleshooting | 86 |
| 8.1 | CMake not found | 86 |
| 8.2 | Notepad++ Does Not Open During Workspace Creation | 87 |
| 8.3 | Some Breakpoints Not Being Hit | 88 |
| 8.4 | (SymbolInfo.dll) The *.die File Is Too Large to Load | 88 |
| 8.5 | Windows Cannot Access Localhost While Using Sync Attribute in EEPROM | 88 |
| 8.6 | Redirecting Function Calls Did Not Work as Expected | 94 |
| 8.7 | License Check Failed | 94 |
| 8.8 | Building Sources Failed | 95 |
| 8.9 | Indentation Errors in YAML File | 95 |
| 8.10 | Failed to Parse Symbols | 96 |
| 8.11 | Could not Load the vECU Binary | 97 |
| 8.12 | Skipping Plugin | 99 |
| 9 | Contact Information | 100 |

1 Introduction

In this chapter you can find information about the intended use, the addressed target group and information about safety and privacy related topics.

1.1 Intended Use

The product is designed to produce a virtual ECU for microcontrollers from existing ECU source code or from precompiled binaries. The virtual ECU is designed for simulation, debugging and pre-calibration of ECU software in a computer-based virtual simulation environment.

In general, virtual ECUs may not be real-time capable. If you control physical devices with a virtual ECU, the system may respond unexpectedly. Take suitable precautions to ensure safe operation.

ETAS GmbH cannot be made liable for damage which is caused by incorrect use and not adhering to the safety information. Please adhere to the ETAS Safety Advice (see `documentation` folder).

1.2 Target Group

This product is directed at trained qualified personnel in development of automotive ECU software (e.g., function developer, application engineer, ECU software integrator, system engineer or calibration engineer) at OEMs, tier-1 or tier-2 suppliers in the auto-motive industry. Technical knowledge in control unit engineering is a prerequisite. In addition, programming knowledge in C/C++ is required. AUTOSAR Classic knowledge is helpful.

1.3 Data Protection

If the product contains functions that process personal data, legal requirements of data protection and data privacy laws shall be complied with by the customer. As the data controller, the customer usually designs subsequent processing. Therefore, he must check if the protective measures are sufficient.

1.4 Data and Information Security

To securely handle data in the context of this product, see the next sections about data and storage locations as well as technical and organizational measures.

1.4.1 Data and Storage Locations

The following sections give information about data and their respective storage locations for various use cases.

License Management

When using the ETAS License Manager in combination with user-based licenses that are managed on the FNP license server within the customer's network, the following data are stored for license management purposes:

Data

- Communication data: IP address
- User data: Windows user ID

Storage location

- FNP license server log files on the customer network

When using the ETAS License Manager in combination with host-based licenses that are provided as FNE machine-based licenses, the following data are stored for license management purposes:

Data

- Activation data: Activation ID
 - Used only for license activation, but not continuously during license usage

Storage location

- FNE trusted storage
`C:\ProgramData\ETAS\FlexNet\fne\license\ts`

1.4.2 Technical and Organizational Measures

We recommend that your IT department takes appropriate technical and organizational measures, such as classic theft protection and access protection to hardware and software.

2 About VECU-BUILDER

VECU-BUILDER is designed to build a virtual ECU (vECU). The vECU can be used for simulation, debugging and pre-calibration of ECU software in a computer-based virtual simulation environment.

VECU-BUILDER supports the generation of Level-1, Level-2, and Level-3 vECUs according to the [Prostep Definition](#) of vECUs. Level-4 vECUs, i.e., hex-files for a specific target, are not supported.

VECU-BUILDER is based on Python and CMake. The inputs can either be C/C++ source codes or binaries like object files or shared libraries including symbol information. In contrast to AUTOSAR Classic, the configuration of a vECU is done in a single YAML file (`vEcuConf.yaml`). No ARXML files are processed. The properties are configured in this text-based file. This file is used to define the supported features of the vECU such as an XCP slave or initial data as part of simulated NVRAM. VECU-BUILDER wraps the binaries of the vECU into an FMU (FMI 2.0 for Co-Simulation). These FMUs can be integrated into any FMI-compliant simulation master.

2.1 VECU-BUILDER on YouTube

A playlist about VECU-BUILDER and its features is available on YouTube. To open the playlist, click on the image below.



2.2 Warning and Error Messages

VECU-BUILDER may encounter situations in which an Error or a Warning message is displayed.

Errors are printed in red and indicate a severe issue which prevents the build from succeeding.

```
[09:58:48] 3 of 4: Importing files and folders to "vECU/imported"
- Extracting "C:/ProgramData/ETAS/VECU-BUILDER/Examples_.../BCU/SilExportBCU.zip"
- Copying contents of "C:/ProgramData/ETAS/VECU-BUILDER/Examples_.../BCU/additonal_sources"
File/folder "C:/ProgramData/ETAS/VECU-BUILDER/Examples_.../BCU/build/additional_scripts"
defined in import_into_project section of vEcuConf.yaml not found.
*** FAILURE ***
```

Fig. 2-1: Error message

Warnings are printed in yellow and are meant to draw the attention to a certain issue during the build. The issue is not as severe as an error and thus the build continues.

```
### Importing files and folders ###
#####
[08:07:34] 1 of 4: Reading config: vEcuConf.yaml
[08:07:34] 2 of 4: Running scripts triggered through "before_import"
Script "../vECU/imported/6_get_characteristics.py" defined in additional_scripts section of vEcuConf.yaml
not found.
All additional scripts will be ignored.
```

Fig. 2-2: Warning message

2.3 Basics

The basic principle is to keep the data lean in a simple and smart way. The concept is the simplification of the ECU software stack and the ARXML file. The A2L file is patched by removing all hardware dependencies and updating memory addresses of all inputs, outputs, measurements, and characteristics. The software stack layers are represented by C and H files which are reflected in the imported folder (vECU\imported) in the vECU build process. The result is a stand-alone FMU containing the model description (e.g. its variables) as XML file, the access to calibration and measurement variables via patched A2L file and an executable model as DLL/SO file.

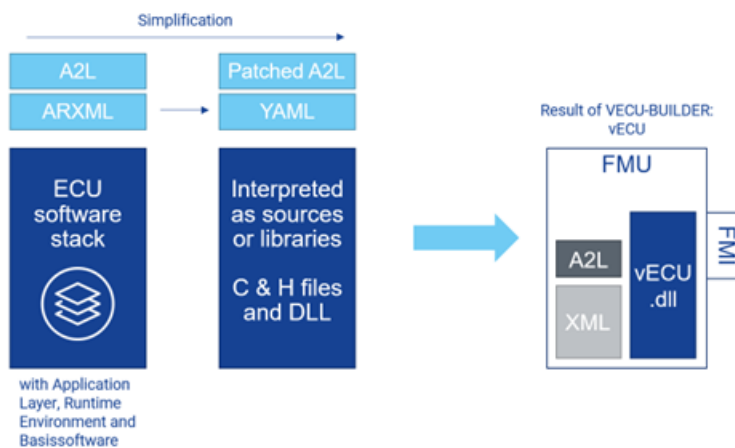


Fig. 2-3: Basic concept and result of VECU-BUILDER

2.4 Virtual ECU

A vECU is a virtualized ECU which can be used as a real ECU. With the vECU you can test the ECU software and execute the software functionality without hardware. This gives you the possibility to test the communication between the ECUs before prototypes or hardware is available. The vECU contains the code, the parameters and the XCP slave as an alternative path to the hex code.

2.5 vECU Creation Process Workflow

The whole workflow is an iterative process to get to the final configuration of the YAML file. The listed points give a rough overview of the workflow. Section A and F are taking place out of the VECU-BUILDER.

- A. Prepare sources
 - Directives that refer to header files in code must be fixed
 - Generate a script collecting the files you need from the various locations you found
- B. Compile sources, incompatible sources must be removed
 - Generate new workspace
 - Copy sources into workspace
 - Build
 - Check error messages
 - Remove or patch code
- C. Link sources and create stubs
 - Solve link errors with empty stubs
- D. Define Inputs and Outputs (I/O) to make the vECU runnable
 - Use symbol information to generate I/O
 - Manually patch the sources of virtual devices
 - Use the C notation of the variables (e.g., `sensor.*`)
- E. Create task model to run the tasks
 - Use text format to define task model
- F. Operate for first time, apply SiL specific code changes
 - Debug code
 - Fill some stub functions with code or apply SiL specific code changes

After building the first iteration of an vECU it can be used to perform further steps like (out of VECU-BUILDER):

- Integrate vECU with plant models and execute it in Co-Simulation-environment
- Run and test the vECU in an experiment environment
- Measurement and calibration of vECU
- Debugging with source code editor

2.6 Functional Mock-up Interface (FMI)

The Functional Mock-up Interface is a free specification that defines a container and interface for exchanging dynamic simulation models. VECU-BUILDER supports Co-simulation (CS). The FMU is delivered with its own solver in Co-simulation.

More information about FMI can be found [here](#).

Fig. 2-4 shows the general steps for Co-Simulation in FMI version 2 and version 3.

Steps 4-6 are repeated until `fmi2Terminate` or `fmi3Terminate` state is called.

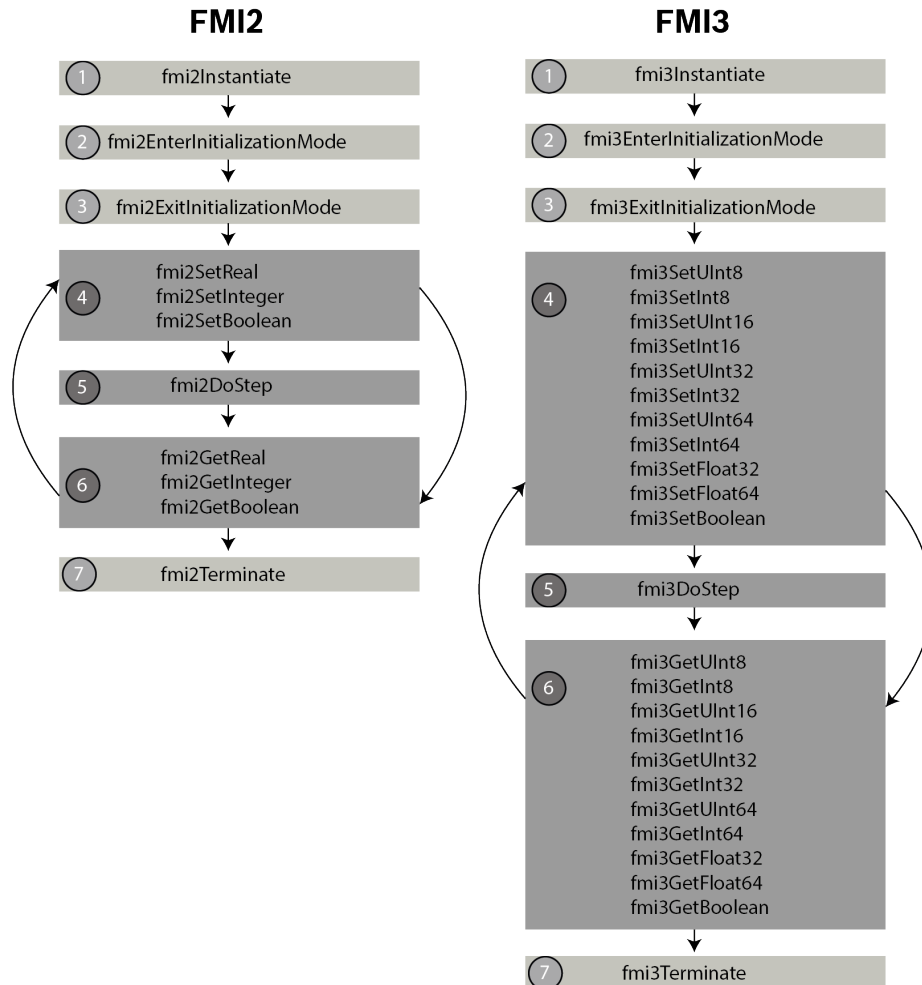


Fig. 2-4: General steps for Co-Simulation

VECU-BUILDER supports FMI2 and FMI3 by a plugin concept. To get more information about plugin concept, see [Template For Plugin V1 \(FMI2\)](#) and [Template For Plugin V2 \(FMI3\)](#).

3 Installation

This chapter provides information for preparing and performing the installation and for licensing the software. The installation can be fulfilled for the following operating systems:

- Windows 10
- Ubuntu 20.04 LTS
- WSL for Windows

3.1 Hardware Requirements

The following Hardware Requirements need to be met:

| | |
|-----------------|---|
| Processor | min. 2 GHz 3 GHz Dual-Core or higher recommended |
| Memory | min. 8 GB RAM 32 GB RAM recommended |
| Free Disk Space | 5 GB (not including the size for application data) >100 GB recommended |

3.2 Preparation

Prior to the installation, check that your computer meets the Hardware and Software Requirements. Depending on the operating system used and network connection, you must ensure that you have the required user rights.



Note

Ensure that you have the necessary access privileges for the installation of the software. If in doubt, contact your system administrator.

3.3 Installation Content

The installation content can be downloaded from [ETAS license and download portal](#) by login via your E-mail address and then be installed.

It contains information about the open-source software attributions, important information like Safety Advice or the User Guide and the executable installation files.



Note

If the download files or download link are not available, contact [Technical Support](#).

3.4 Licensing

A valid license is required to use the software. You can obtain a license in one of the following ways:

- from your tool coordinator
- via the self-service portal on the ETAS website at www.etas.com/support/licensing
- via the ETAS License Manager

To activate the license, you must enter the Activation ID that you received from ETAS during the ordering process.

For more information about ETAS license management, see the [ETAS License Management FAQ](#) or the ETAS License Manager help.

To open the ETAS License Manager help

The ETAS License Manager is available on your computer after the installation of any ETAS software.

1. From the Windows Start menu, select **E > ETAS > ETAS License Manager**.
The ETAS License Manager opens.
2. Click in the ETAS License Manager window and press F1.
The ETAS License Manager help opens.

VECU-BUILDER checks

- the product license when building FMUs.
- the run-time license during run-time of the vECU.
- the XCP license before establishing an XCP connection.
- the GO license during build-time. If it is valid, it will prevent all license checks during run-time.

3.5 Installation on Windows 10

3.5.1 Software Requirements for Windows 10

The following Software Requirements need to be met:

| | |
|----------------------|--|
| Required Software | ETAS License Manager CMake (version ≥ 3.15) |
| Recommended Software | Notepad++ |
| Optional Software | Visual Studio 2015, 2017, 2019, 2022 Visual Studio Code Python |

3.5.2 Manual Installation of VECU-BUILDER

1. Go to the directory where the installation file is located and execute the `VECU_BUILDER_installer_1.6.0.exe` file.
 - ⇒ The Setup Wizard opens.
2. Click **Next**.
 - ⇒ The "Safety Advice" window opens.
3. Read the Safety Advice carefully, then select "I read and accept the Safety Advice".
4. Click **Next**.
 - ⇒ The "Installation Path" window opens.
5. Accept the default path (click **Next**) or click **Browse** to select a custom location.
 - ⇒ The "Ready to Install" window opens.
6. Click **Install**.
 - ⇒ The installation is performed, its progress is shown via a progress bar.
7. Click **Next**.
 - ⇒ The "Third-party Software" window opens.
8. Install CMake (required) and Notepad++ (recommended).
See the links below in the installation dialog:
 - [CMake](#) (version 3.15 or higher)
 - [Notepad++](#)
9. Click **Next**.
 - ⇒ The "Completing VECU-BUILDER Setup" window opens.
10. Optionally, activate the Open VECU-BUILDER documentation checkbox to open the documentation folder.
11. Click Finish.
 - ⇒ The installation is completed, and the VECU-BUILDER can now be used.

3.5.3 Silent Installation of VECU-BUILDER

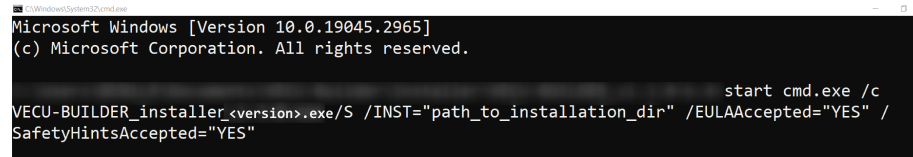
Besides the Manual Installation, you also can use the Silent Installation. Installation differs between using the Command Prompt and the PowerShell.

Silent Installation using Command Prompt

1. Open the command prompt.
2. Navigate to the directory where the installer (VECU-BUILDER_installer_1.6.0.exe) is located.
3. Execute the following command:


```
start cmd.exe /c VECU-BUILDER_installer_1.6.0.exe /S
/INST="path_to_installation_dir" /EULAAccepted="YES"
/SafetyHintsAccepted="YES"
```

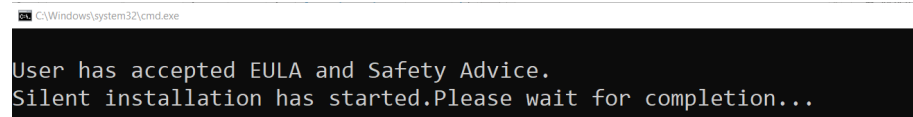
where `path_to_installation_dir` contains a path to a directory where the software is to be installed.



```
Microsoft Windows [Version 10.0.19045.2965]
(c) Microsoft Corporation. All rights reserved.

start cmd.exe /c
VECU-BUILDER_installer_<version>.exe/S /INST="path_to_installation_dir" /EULAAccepted="YES" /
SafetyHintsAccepted="YES"
```

- ⇨ A new command prompt window opens and installation starts.



```
C:\Windows\system32\cmd.exe

User has accepted EULA and Safety Advice.
Silent installation has started.Please wait for completion...
```

Silent Installation using PowerShell

1. Open the PowerShell.
2. Navigate to the directory where the installer (VECU-BUILDER_installer_1.6.0.exe) is located.
3. Execute the following command:

```
Start-Process -FilePath ".\VECU-BUILDER_installer_1.6.0.exe/" -ArgumentList "/c /S /INST= path_to_installation_dir /EULAAccepted=YES /SafetyHintsAccepted=YES" -Wait
```

 where `path_to_installation_dir` contains a path to a directory where the software is to be installed.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Windows\system32> Set-Location .\
PS C:\Users\... \Documents> Start-Process -FilePath ".\VECU-BUILDER_installer_<version>.exe/" -ArgumentList "/c /S /INST=C:\Program Files\ETAS\VECU-BUILDER /EULAAccepted=YES /SafetyHintsAccepted=YES" -Wait
>>
```

Or

4. Execute the following command:

```
Start-Process -FilePath " path_to \VECU-BUILDER_installer_1.6.0.exe/" -ArgumentList "/c /S /INST= path_to_installation_dir /EULAAccepted=YES /SafetyHintsAccepted=YES" -Wait
```

 where `path_to` contains the path where the installer (VECU-BUILDER_installer_1.6.0.exe) is located and `path_to_installation_dir` contains a path to a directory where the software is to be installed.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Windows\system32> Start-Process -FilePath ".\VECU-BUILDER_installer_<version>.exe/" -ArgumentList "/c /S /INST= path_to_installation_dir /EULAAccepted=YES /SafetyHintsAccepted=YES" -Wait
>>
```

⇒ Installation starts.

```
User has accepted EULA and Safety Advice.
Silent installation has started.Please wait for completion...
```

3.5.4 Uninstalling VECU-BUILDER on Windows 10

1. Open the location where you installed VECU-BUILDER.
 If you used the default installation location, you can find it under
`C:/Program Files/ETAS/VECU-BUILDER`
2. Execute the `uninstall.exe` with double-click.

3.6 Installation on Ubuntu 20.04 LTS

3.6.1 Software Requirements for Ubuntu 20.04 LTS

The following Software Requirements need to be met:

| | |
|-------------------|---|
| Required Software | ETAS License Manager cmake build-essential gcc-multilib g++-multilib libssl-dev:i386 linux-libc-dev:i386 xterm |
| Optional Software | Visual Studio Code Python nano |

3.6.2 Installing License Manager (LiMa) on Ubuntu 20.04 LTS

In order to use VECU-BUILDER, you need to install **ETAS License Manager** manually prior to installation of VECU-BUILDER.

The installation debian packages for LiMa are delivered next to the VECU-BUILDER installation debian package.

| |
|-----------------------------|
| LiMa-1.8.11.24-Linux.deb |
| LiMaX64-1.8.11.24-Linux.deb |

Fig. 3-1: LiMa installation debian packages

1. Navigate to the directory where the LiMa debian Package files are located.
 2. Install LiMa using the following command:

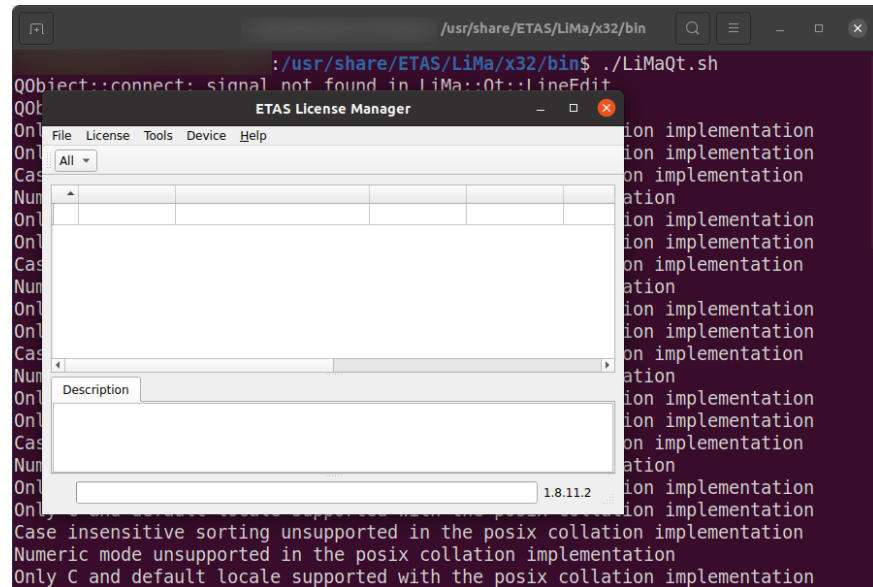

```
sudo apt install ./LiMa-1.8.11.24-Linux.deb
```
 3. Install LiMaX64 using the following command:


```
sudo apt install ./LiMaX64-1.8.11.24-Linux.deb
```
 4. Fix broken installs using the following command:


```
sudo apt --fix-broken install
```
- ⇒ LiMa and all its components are installed.

3.6.3 Opening ETAS License Manager on Ubuntu 20.04 LTS

1. Navigate to the following direction:
`/usr/share/ETAS/LiMa/x32/bin`
 2. Open a new terminal.
 3. Enter the following command:
`./LiMaQt.sh`
- ⇒ LiMa was opened.



3.6.4 Activating the LiMa License

There are several possibilities to activate the license. For more information about ETAS license management, see the [ETAS License Management FAQ](#) or the ETAS License Manager help.

3.6.5 Installing VECU-BUILDER on Ubuntu 20.04 LTS

1. Navigate to the directory where the Debian Software Package file `VECU-BUILDER_installer_1.6.0.deb` is located.
2. Open a new terminal.
3. Execute the following command:

```
sudo apt install ./VECU-BUILDER_installer_1.6.0.deb
```

Note

VECU-BUILDER has dependencies on other software. The dependent software packages will be installed during the installation. An Internet connection is required to install the dependent software packages.

4. Accept the installation of dependent packages.
 - ⇒ The packages are selected and unpacked.
5. Accept the Safety Advice.
 - ⇒ The VECU-BUILDER package deployment is completed.
6. Logout and login to enable environment variables to be set.

3.6.6 Uninstalling VECU-BUILDER on Ubuntu 20.04 LTS

1. Open a new terminal.
2. Execute the following command:

```
sudo apt remove vecu-builder
```

 - ⇒ You are asked if you want to continue uninstalling.
3. To continue, enter `y` and hit **Enter**.
 - ⇒ The VECU-BUILDER package is removed.

3.7 Installation on Ubuntu 20.04 for WSL

It is possible to create a Linux-vECU from a Windows host. To be able to create a Linux-vECU from a Windows host, the following prerequisites must be met:

- WSL is installed on Windows.
- Ubuntu 20.04 is installed on WSL.
- LiMa is installed on WSL.
- VECU-BUILDER is installed on WSL.

3.7.1 Software Requirements for Ubuntu 20.04 on WSL

The following Software Requirements need to be met:

| | |
|-------------------|---|
| Required Software | ETAS License Manager cmake build-essential gcc-multilib g++-multilib libssl-dev:i386 linux-libc-dev:i386 gnome-terminal (for dialog mode applications) |
| Optional Software | Visual Studio Code (for debugging, installed on Windows PC Host) Python nano gdb (for debugging) |

3.7.2 Installing WSL on Windows

To install WSL on Windows, see [Install WSL command](#).

3.7.3 Installing Ubuntu 20.04 on WSL

1. Open PowerShell.
2. Check what distributions are available online in PowerShell using the following command:

```
wsl --list --online
```

3. Install Ubuntu 20.04 on WSL using the following command:

```
wsl --install -d ubuntu-20.04
```

- ⇒ Ubuntu 20.04 is installed.

3.7.4 Installing Dependent Software Packages

In order to install VECU-BUILDER you need to install dependent software packages.



Note

The installation of depended software packages only works if you have unrestricted access to the internet. The `sudo apt` commands will fail if your computer is not allowed to connect your Linux to the official package repositories. In this case, ask your IT department for help.



Note

Downloading dependencies or installing VECU-BUILDER only runs in WSL1, using VECU-BUILDER only runs in WSL2. Make sure that the WSL version matches the respective action. If necessary, you need to change the version.

For WSL1: `wsl --set-version Ubuntu-20.04 1`

For WSL2: `wsl --set-version Ubuntu-20.04 2`

1. In PowerShell, check the WSL Ubuntu version using the following command:

```
wsl -l -v
```

If it is not 1, set the version to 1, using the following command:

```
wsl --set-version Ubuntu-20.04 1
```

2. Open Ubuntu 20.04 command line interface.
3. Install the `i386` architecture using the following command:

```
sudo dpkg --add-architecture i386
```

4. Install `libc6-i386` using the following command:

```
sudo apt install -y libc6-i386
```

5. Install `lsb` using the following command:

```
sudo apt install -y lsb
```

6. Run a package update using the following command:

```
sudo apt update
```

7. Run a package upgrade using the following command:

```
sudo apt upgrade
```

8. Install `gnome-terminal` using the following command:

```
sudo apt install gnome-terminal
```

You need `gnome-terminal` for debugging, ETAS License Manager (LiMa) and the Dialog mode.)

9. Install `gdb` using the following command:

```
sudo apt install gdb
```

You need `gdb` for debugging.

3.7.5 Installing License Manager (LiMa) on Ubuntu 20.04 for WSL

In order to use VECU-BUILDER, you need to install **ETAS License Manager** manually prior to installation of VECU-BUILDER.

The installation debian packages for LiMa are delivered next to the VECU-BUILDER installation debian package.

| |
|-----------------------------|
| LiMa-1.8.11.24-Linux.deb |
| LiMaX64-1.8.11.24-Linux.deb |

Fig. 3-2: LiMa installation debian packages

1. In order to install LiMa copy the required installation debian packages for LiMa to user home on Ubuntu.
2. Open Ubuntu 20.04 command line interface.
3. Install LiMa using the following command:

```
sudo apt install ./LiMa-1.8.11.24-Linux.deb
```

4. Install LiMaX64 using the following command:


```
sudo apt install ./LiMaX64-1.8.11.24-Linux.deb
```

5. Fix broken installs using the following command:

```
sudo apt --fix-broken install
```

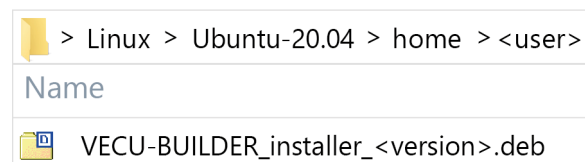
⇒ LiMa and all its components are installed.

3.7.6 Installing VECU-BUILDER on Ubuntu 20.04 for WSL

 **Note**

The installation of VECU-BUILDER only works if you have unrestricted access to the internet. The `sudo apt` commands will fail if your computer is not allowed to connect your Linux to the official package repositories. In this case, ask your IT department for help.

1. In order to install VECU-BUILDER copy VECU-BUILDER_installer_1.6.0.deb to user home on Ubuntu.



2. Install VECU-BUILDER using the following command:

```
sudo apt install ./VECU-BUILDER_installer_1.6.0.deb
```

Note

VECU-BUILDER has dependencies on other software. The dependent software packages will be installed during the installation. An Internet connection is required to install the dependent software packages.

3. Close and restart Ubuntu 20.04.

To create a workspace using WSL Ubuntu 20.04, see [Ubuntu 20.04 on WSL Command Line Interface](#).

3.7.6.1 Opening ETAS License Manager on Ubuntu 20.04 LTS for WSL

Prerequisites:

- Make sure to have the latest Ubuntu 20.04 version and all related packages installed.
- Make sure to have XTerm (Ubuntu-20.04) installed.
- Make sure to have UXTerm (Ubuntu-20.04) installed.

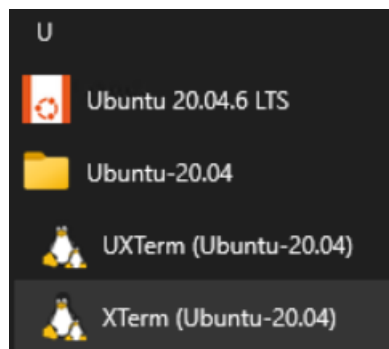


Fig. 3-3: UXTerm and XTerm for Ubuntu 20.04

More information about running Linux GUI apps on the Windows Subsystem for Linux can be found [here](#).

1. Open PowerShell and set WSL Ubuntu 20.04 version to 2 using the following command:

```
wsl --set-version Ubuntu-20.04 2
```

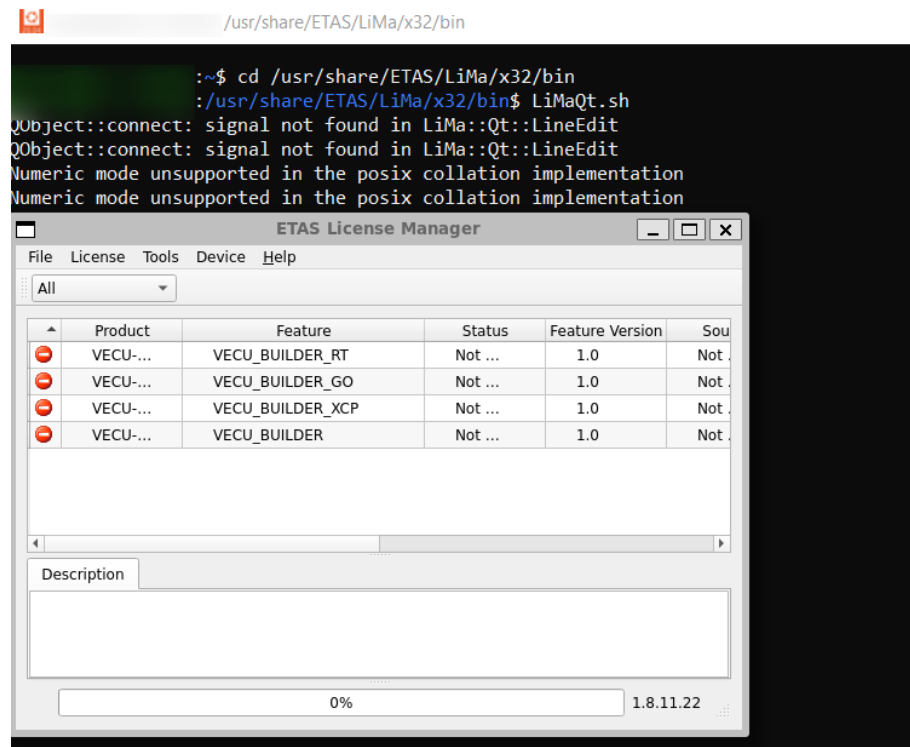
2. Open Ubuntu 20.04 and change directory using the following command:

```
cd /usr/share/ETAS/LiMa/x32/bin
```

3. Open `LiMaQt.sh` using the following command:

```
LiMaQt.sh
```

- ⇒ LiMa was opened.



3.7.7 Activating the LiMa License

There are several possibilities to activate the license. For more information about ETAS license management, see the [ETAS License Management FAQ](#) or the ETAS License Manager help.

3.7.8 Uninstalling VECU-BUILDER on Ubuntu 20.04 for WSL

1. Open Ubuntu 20.04 command line interface.
2. Execute the following command:


```
sudo apt remove vecu-builder
```

⇒ You are asked if you want to continue uninstalling.
3. To continue, enter `Y` and hit **Enter**.

⇒ The VECU-BUILDER package is removed.

3.8 Installed Files and Folders

VECU-BUILDER Tool

The default installation location is

C:/Program Files/ETAS/VECU-BUILDER/1.6.0 on **Windows**

Or

/opt/etas/VECU-BUILDER/1.6.0 on **Ubuntu 20.04 LTS** and **Ubuntu 20.04 on WSL**.

It is recommended not to alter the installation location.

An environment variable of `VECU_BUILDER_HOME` points to this folder.

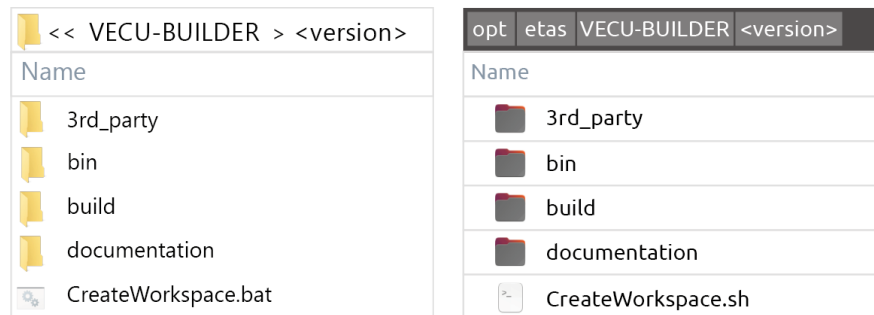


Fig. 3-4: Installation content (left: Windows, right: Ubuntu 20.04 LTS)

The content of this folder consists of several subfolders and one command/shell script:

- `3rd_party`: Contains the third party tools of FMPy and MinGW.
- `bin`: Contains library and execution files for the build process. These files are important for the build and must not be altered.
- `build`: Contains templates, resources, and scripts for the build process. These files are important for the build and must not be altered.
- `documentation`: Contains the VECU-BUILDER User Guide, the OSS Attribution and the ETAS Safety Advice documents.
- `CreateWorkspace.bat` (Windows) / `CreateWorkspace.sh` (Ubuntu 20.04 LTS): Creates a new workspace. After executing, you will be guided through the process step by step.

VECU-BUILDER Examples/Templates

You can find ready-to-use examples in the following location:

C:/ProgramData/ETAS/VECU-BUILDER/Examples_1.6.0 on **Windows**

Or

/opt/etas/VECU-BUILDER/Examples_1.6.0 on **Ubuntu 20.04 LTS** and **Ubuntu 20.04 on WSL**.

An environment variable of `VECU_BUILDER_EXAMPLES` points to this folder.

The following examples are delivered along with the tool:

- BCU (Body Control Unit) only for Windows
- EventTriggerExample:
- SimpleExample

The following templates are delivered along with the tool:

- plugin_template_v1_FMI2
- plugin_template_v2_FMI3

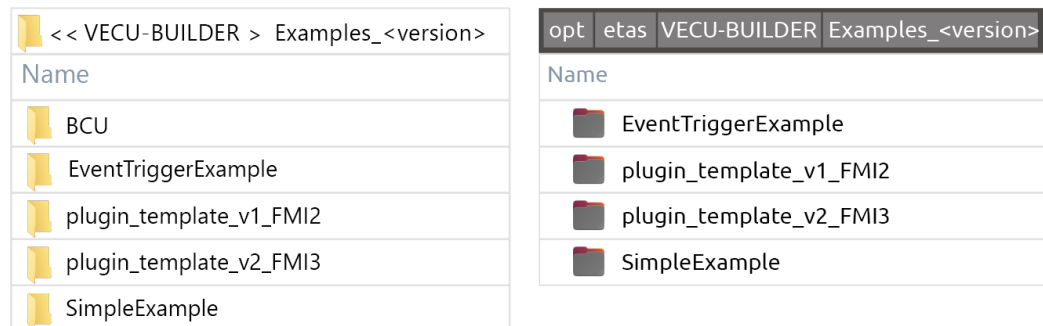


Fig. 3-5: Delivered examples/templates (left: Windows, right: Ubuntu 20.04 LTS)

VECU-BUILDER Workspaces

As location for all your workspaces we recommend the default folder, where you should create a dedicated subfolder for each workspace.

The default folder is created during the installation process on **Windows** under

C:/Users/Public/Documents/VECU-BUILDER_Workspaces

Or

/opt/etas/VECU-BUILDER_Workspaces on **Ubuntu 20.04 LTS** and **Ubuntu 20.04 on WSL**.

Access to Artefacts in Windows

You can access all artefacts in Windows via their respective Start Menu entries.

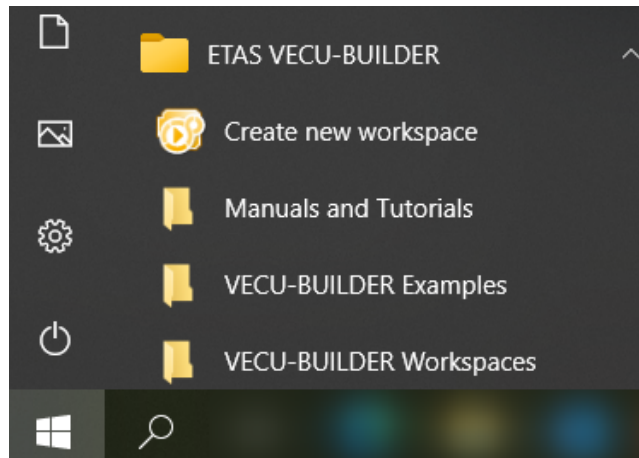


Fig. 3-6: Start Menu entries

4 Working with VECU-BUILDER

To commence your learning, we recommend following the bellow path:

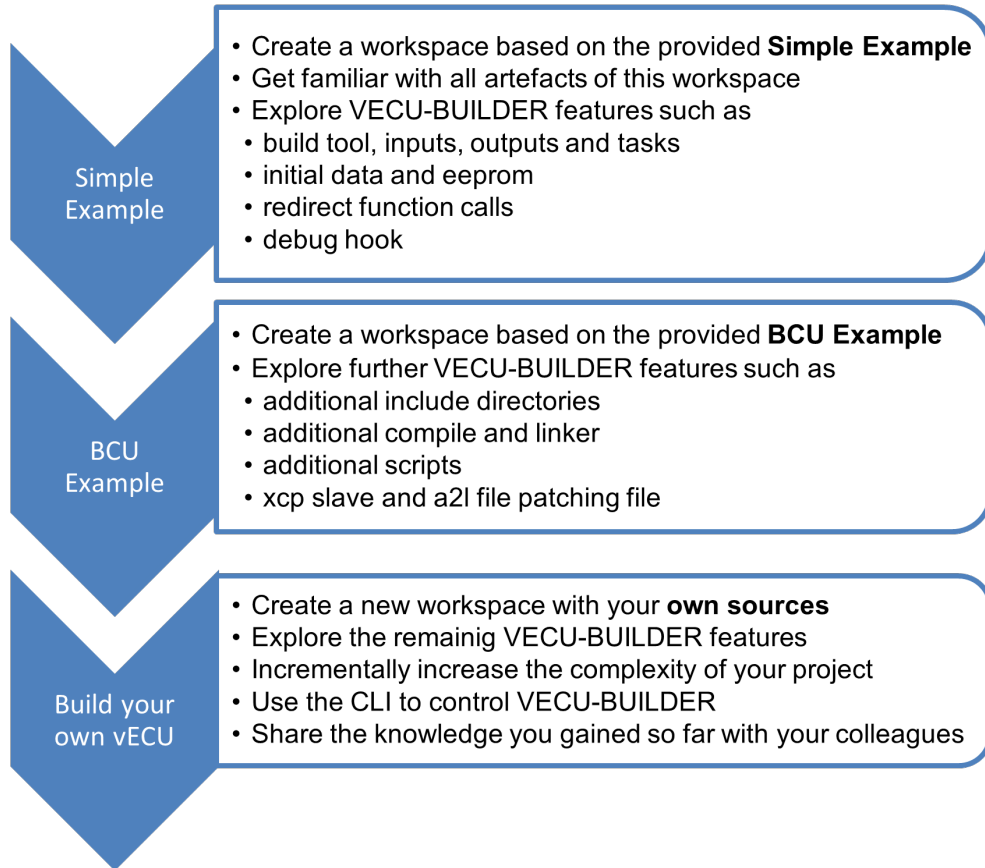


Fig. 4-1: The learning path

This section guides you through the process of creating a vECU in four distinct stages. Each stage can be triggered individually, and you can choose to continue with the next one.



Fig. 4-2: VECU-BUILDER stages

By following the steps described in the next chapters, you will build your first vECU based on the [Simple Example](#). This is the ideal starting point for your virtualization leaning journey.

4.1 Creating a New Workspace

The very first step, required at the beginning of every project, is to create a workspace.



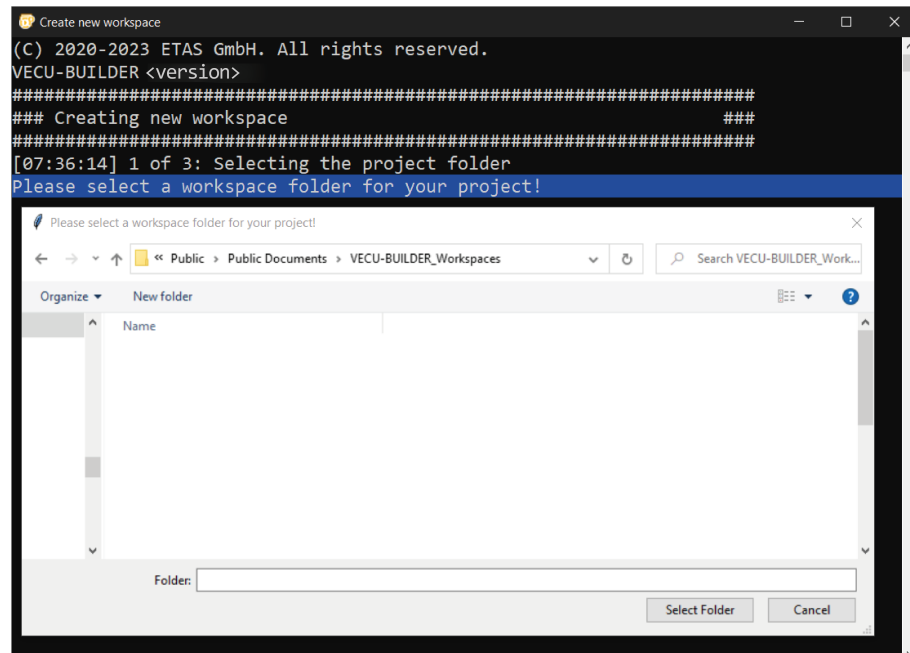
Note

Workspaces are designed for parallel use.

A single workspace cannot be used for tasks running in parallel.

4.1.1 Creating a Workspace on Windows

1. Launch "Create new workspace" from the Start Menu.
 - ⇒ A console window opens providing details on the overall process, various stages it goes through and their individual steps.
 - ⇒ In the first step of "Create new workspace" you will be asked to select a folder where your workspace will be saved.



2. Navigate to the default location of your workspaces
`C:/Users/Public/Documents\VECU-BUILDER_Workspaces`
 and select an existing folder or create a new one.

- ⇒ The configuration file `vEcuConf.yaml` opens in Notepad++.
- ⇒ Per default, this is the configuration file of the [Simple Example](#).

The screenshot shows two windows. The top window is a terminal titled 'Create new workspace' with the following text:

```
(C) 2020-2023 ETAS GmbH. All rights reserved.
VECU-BUILDER <version>
#####
### Creating new workspace ###
#####
[07:36:14] 1 of 3: Selecting the project folder
Please select a workspace folder for your project!
New project path is "C:/Users/Public/Documents/VECU-BUILDER_Workspaces/SimpleExample"
[07:37:42] 2 of 3: Creating project template
[07:37:42] 3 of 3: Please update "vEcuConf.yaml"
Please save and close the editor after editing "vEcuConf.yaml"
```

The bottom window is Notepad++ editing `vEcuConf.yaml`. The visible content is:

```
1 #####
2 # The version of the .yaml file schema #
3 #####
4 version:
5
6 #####
7 # build_sources or import_compiled #
8 # build_sources: You import sources, header files, static libraries and let #
9 # prvEcuBuild compile, link and build a dll. #
10 # The dll will be named <fmu_name>.dll. #
11 # import_compiled: Just import a dll, wrap it with a fmu wrapper, setup the #
12 # source outputs and packs #
```

3. Keep the configuration file as is and close the Notepad ++ application.
- ⇒ Your new workspace is now created.

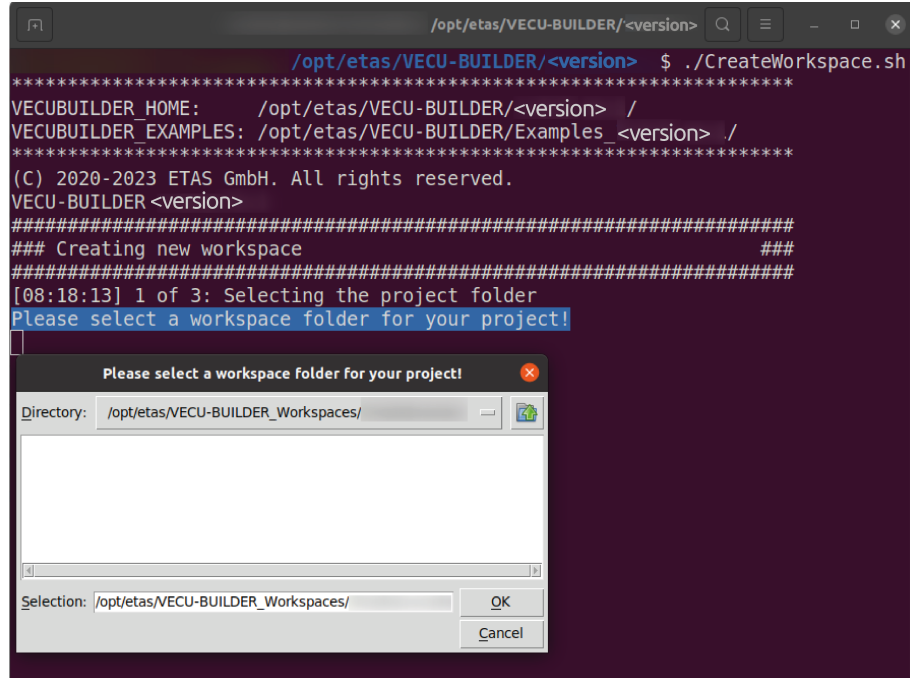
The process will automatically continue with the next stage.

4.1.2 Creating a Workspace on Ubuntu 20.04 LTS

Note

In Ubuntu 20.04 LTS the folder, that should be used as workspace, needs to exist before the workspace creation is proceeded.

1. Navigate to the folder, where the `CreateWorkspace.sh` is located. The default path is `opt/etas/VECU-BUILDER/1.6.0`.
2. Open a new terminal. VECU-BUILDER will use the editor found under `/usr/bin/editor`.
3. Enter `./CreateWorkspace.sh`.
- ⇒ In the first step of "Create new workspace" you will be asked to select a folder where your workspace will be saved.
4. Navigate to the default location of your workspaces `/opt/etas/VECU-BUILDER_Workspaces` and select an existing folder.



```

/opt/etas/VECU-BUILDER/<version> $ ./CreateWorkspace.sh
*****
VECUBUILDER_HOME: /opt/etas/VECU-BUILDER/<version> /
VECUBUILDER_EXAMPLES: /opt/etas/VECU-BUILDER/Examples_<version> ./
*****
(C) 2020-2023 ETAS GmbH. All rights reserved.
VECU-BUILDER <version>
#####
### Creating new workspace ###
#####
[08:18:13] 1 of 3: Selecting the project folder
Please select a workspace folder for your project!
  
```

- ⇒ The configuration file `vEcuConf.yaml` opens.
- ⇒ Per default, this is the configuration file of the [Simple Example](#).

```

/opt/etas/VECU-BUILDER/<version> $ ./CreateWorkspace.sh
*****
VECUBUILDER HOME: /opt/etas/VECU-BUILDER/<version> /
VECUBUILDER_EXAMPLES: /opt/etas/VECU-BUILDER/Examples_<version> /
*****
(C) 2020-2023 ETAS GmbH. All rights reserved.
VECU-BUILDER <version>
#####
### Creating new workspace ###
#####
[08:18:13] 1 of 3: Selecting the project folder
Please select a workspace folder for your project!
New project path is "/opt/etas/VECU-BUILDER Workspaces/SimpleExample".
[08:19:24] 2 of 3: Creating project template
[08:19:24] 3 of 3: Please update "vEcuConf.yaml"
Please save and close the editor after editing "vEcuConf.yaml"

/opt/etas/VECU-BUILDER Workspaces/SimpleExample/vEcuConf.yaml
#####
# The version of the .yaml file schema #
#####
version:

#####
# build_sources or import_compiled #
# build_sources: You import sources, header files, static libraries and let #
# prvEcuBuild compile, link and build a dll. #
# The dll will be named <fmu_name>.dll. #
# import_compiled: Just import a dll, wrap it with a fmu wrapper, setup the #
# inputs, outputs and tasks. #
#####

```

5. Keep the configuration file as is and close it.
- ⇒ Your new workspace is now created.

The process will automatically continue with the next stage.

4.2 Importing Files and Folders

During this stage, the sources defined in your `vEcuConf.yaml` are copied to the "`vEcu/imported`" folder in your workspace.



Note

During the import stage, files and folders get copied into the workspace. For reasons of portability, it is recommended to create workspaces that are self-contained.

After successful completion of the previous stage [Creating a New Workspace](#) you were forwarded to the next stage [Importing Files and Folders](#) and the process continues.

If you work in an already existing workspace, you can trigger this stage by running `1_Import.bat` on **Windows** or `1_Import.sh` on **Ubuntu 20.04 LTS**.

After successful completion of this stage **Importing Files and Folders** a dialog opens asking you whether you want to continue with the next stage [Building the vECU](#) or inspect the results of this stage.

```

Create new workspace
VECU-BUILDER <version>
#####
### Importing files and folders ###
#####
[07:42:30] 1 of 4: Reading config: vEcuConf.yaml
[07:42:30] 2 of 4: Running scripts triggered through "before_import"
- No script defined in the vEcuConf.yaml file
[07:42:30] 3 of 4: Importing files and folders to "vECU/imported"
- Copying contents of "C:/ProgramData/ETAS/VECU-BUILDER/E
xamples_<version> /SimpleExample/src"
[07:42:30] 4 of 4: Running scripts triggered through "after_import"
- No script defined in the vEcuConf.yaml file
*** SUCCESS ***
Please answer the question!

```

Question

Do you want to proceed building the sources of vECU or inspect the results of the import?

Yes: Build the sources of vECU.
No: Inspect the results of the import.
Cancel: Stop the process here.

Yes No Cancel

Fig. 4-3: Proceed with vECU Build dialog or inspect the results (Windows)

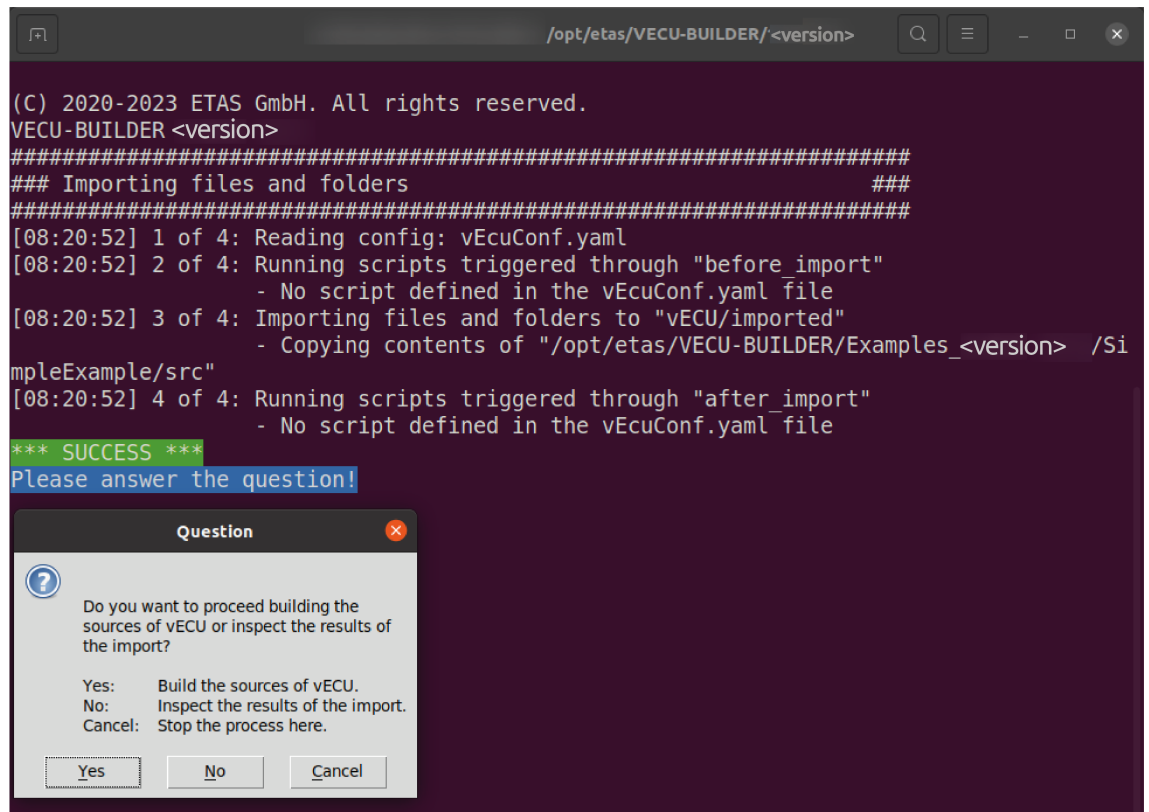


Fig. 4-4: Proceed with vECU Build dialog or inspect the results (Ubuntu 20.04 LTS)

1. Click **Yes**.
- ⇒ Your new workspace is now created.

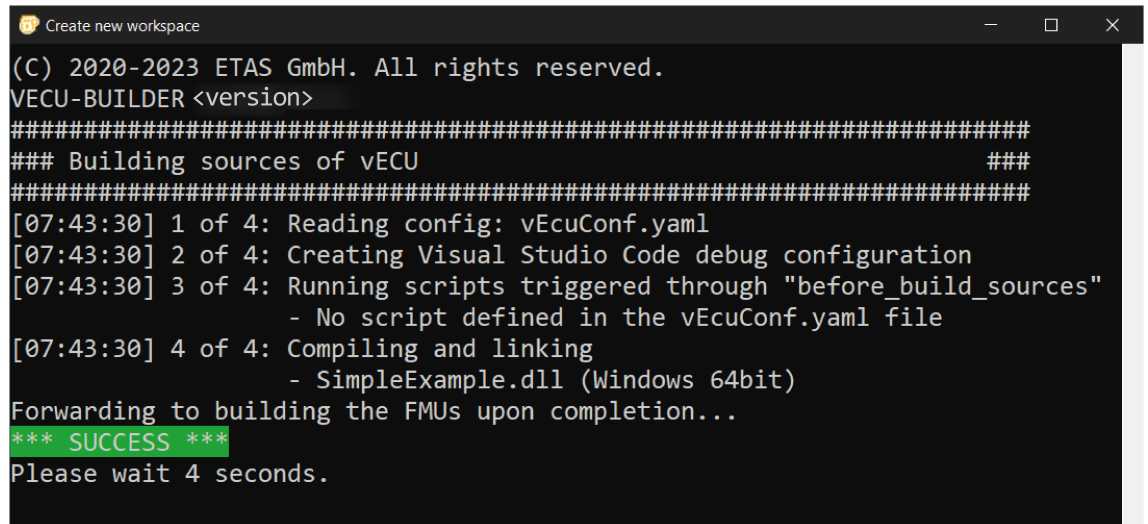
The process will continue with the next stage.

4.3 Building the vECU

During this stage, the sources imported into your workspace are compiled and linked into a DLL/SO file forming the core functionality of your future vECU.

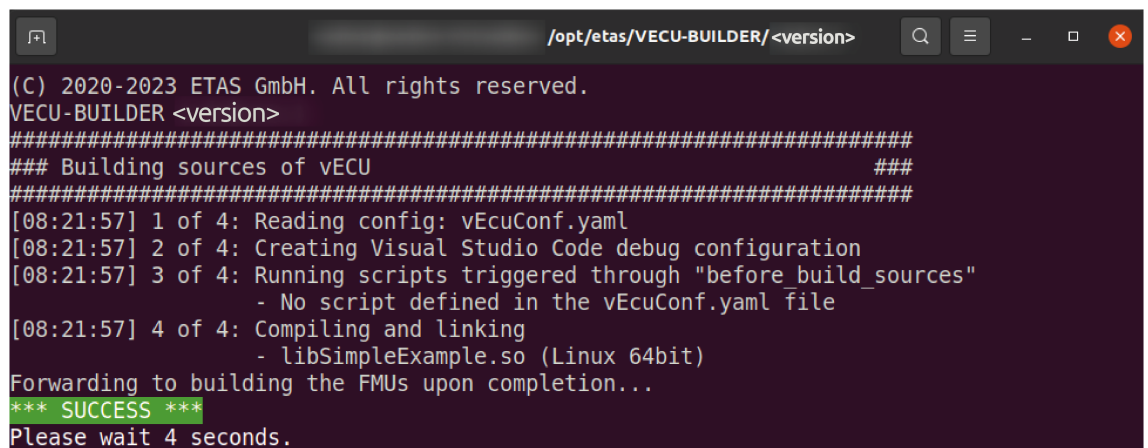
After successful completion of the previous stage [Importing Files and Folders](#) and selecting to proceed with the build of the vECU you were forwarded to the next stage [Building the FMU](#) and the process continues.

If you work in an already existing workspace, you can trigger this stage by running `2_Build.bat` on **Windows** or `2_Build.sh` on **Ubuntu 20.04 LTS**.



```
(C) 2020-2023 ETAS GmbH. All rights reserved.
VECU-BUILDER <version>
#####
### Building sources of vECU ###
#####
[07:43:30] 1 of 4: Reading config: vEcuConf.yaml
[07:43:30] 2 of 4: Creating Visual Studio Code debug configuration
[07:43:30] 3 of 4: Running scripts triggered through "before_build_sources"
- No script defined in the vEcuConf.yaml file
[07:43:30] 4 of 4: Compiling and linking
- SimpleExample.dll (Windows 64bit)
Forwarding to building the FMUs upon completion...
*** SUCCESS ***
Please wait 4 seconds.
```

Fig. 4-5: Building vECU completed (Windows)



```
(C) 2020-2023 ETAS GmbH. All rights reserved.
VECU-BUILDER <version>
#####
### Building sources of vECU ###
#####
[08:21:57] 1 of 4: Reading config: vEcuConf.yaml
[08:21:57] 2 of 4: Creating Visual Studio Code debug configuration
[08:21:57] 3 of 4: Running scripts triggered through "before_build_sources"
- No script defined in the vEcuConf.yaml file
[08:21:57] 4 of 4: Compiling and linking
- libSimpleExample.so (Linux 64bit)
Forwarding to building the FMUs upon completion...
*** SUCCESS ***
Please wait 4 seconds.
```

Fig. 4-6: Building vECU completed (Ubuntu 20.04 LTS)

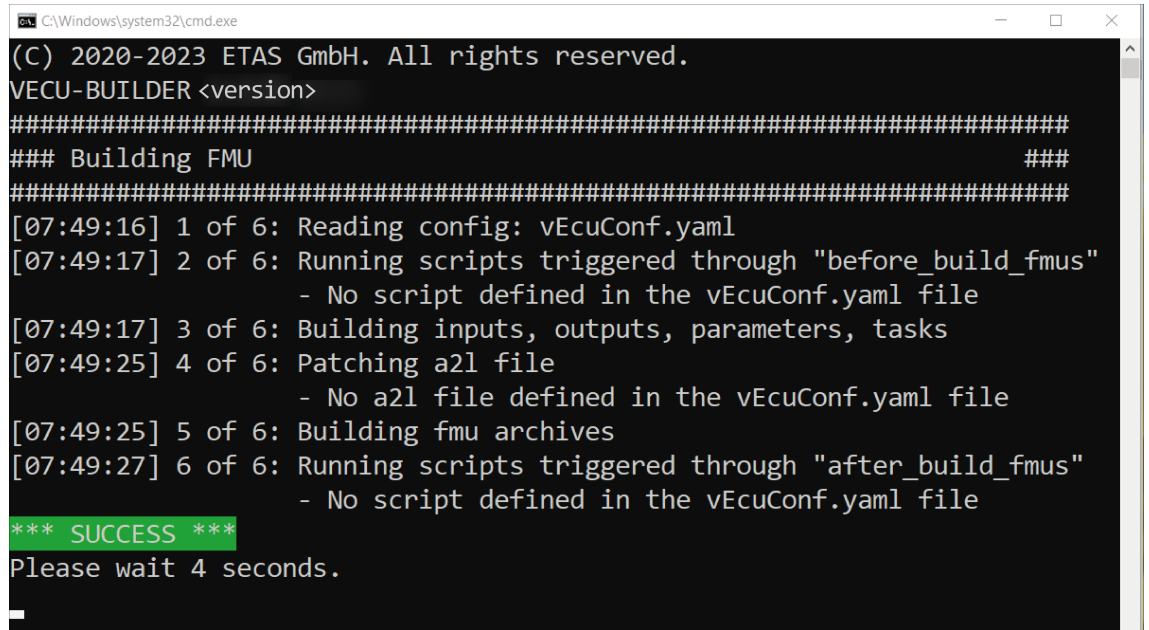
The process will automatically continue with the next stage.

If the process will not automatically continue with the next stage and error messages are displayed, see [Building Sources Failed](#) for more details.

4.4 Building the FMU

During this stage, the DLL/SO file created in the previous stage will be wrapped into an FMU container representing your vECU.

After successful completion of the previous stage [Building the vECU](#) and selecting to proceed with the build of the vECU you were forwarded to the next stage [Building the FMU](#) where the process completes.

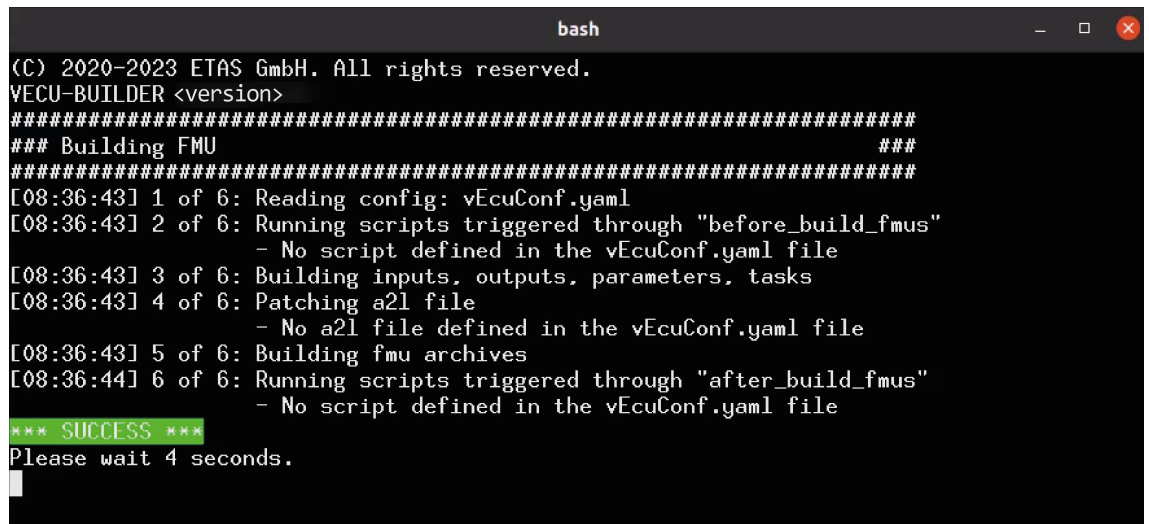


```

C:\Windows\system32\cmd.exe
(C) 2020-2023 ETAS GmbH. All rights reserved.
VECU-BUILDER <version>
#####
### Building FMU                                     ###
#####
[07:49:16] 1 of 6: Reading config: vEcuConf.yaml
[07:49:17] 2 of 6: Running scripts triggered through "before_build_fmus"
                - No script defined in the vEcuConf.yaml file
[07:49:17] 3 of 6: Building inputs, outputs, parameters, tasks
[07:49:25] 4 of 6: Patching a2l file
                - No a2l file defined in the vEcuConf.yaml file
[07:49:25] 5 of 6: Building fmu archives
[07:49:27] 6 of 6: Running scripts triggered through "after_build_fmus"
                - No script defined in the vEcuConf.yaml file
*** SUCCESS ***
Please wait 4 seconds.

```

Fig. 4-7: Building FMU completed (Windows)



```

bash
(C) 2020-2023 ETAS GmbH. All rights reserved.
VECU-BUILDER <version>
#####
### Building FMU                                     ###
#####
[08:36:43] 1 of 6: Reading config: vEcuConf.yaml
[08:36:43] 2 of 6: Running scripts triggered through "before_build_fmus"
                - No script defined in the vEcuConf.yaml file
[08:36:43] 3 of 6: Building inputs, outputs, parameters, tasks
[08:36:43] 4 of 6: Patching a2l file
                - No a2l file defined in the vEcuConf.yaml file
[08:36:43] 5 of 6: Building fmu archives
[08:36:44] 6 of 6: Running scripts triggered through "after_build_fmus"
                - No script defined in the vEcuConf.yaml file
*** SUCCESS ***
Please wait 4 seconds.

```

Fig. 4-8: Building FMU completed (Ubuntu 20.04 LTS)

4.5 Workspace Content

You have now successfully created the VECU-BUILDER workspace and built your first vECU based on the provided Simple Example sources. In this chapter, you find a description of the workspace contents for Windows and Ubuntu 20.04 LTS.

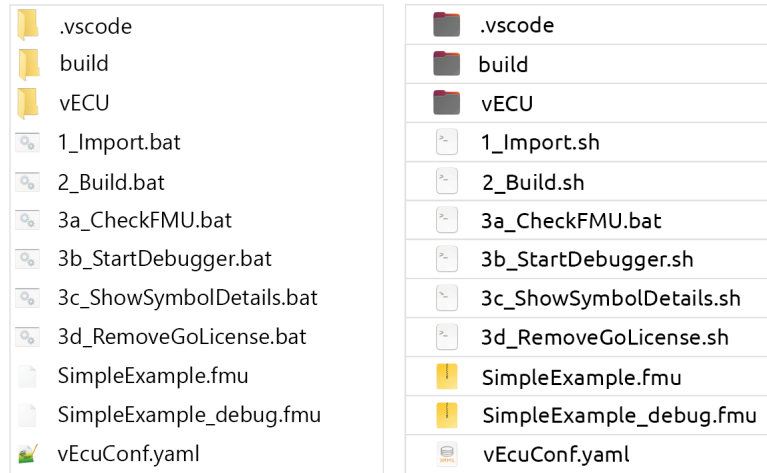


Fig. 4-9: Workspace contents

The content of the workspace consists of several artefacts:

- `vscode` folder:
 - `launch.json` file for vECU debugging in VS Code
- `build` folder:
 - `additional_scripts` folder: location for your project specific additional scripts
 - `log` folder: log files from executed stages
 - `scripts` folder: command and shell scripts to perform the individual stages
 - `last_build_footprint.txt`: details of last performed build stage
 - `RawSymbolDetails.txt`: subset of `SymbolDetails` and for internal purposes only
 - `SymbolDetails.txt`: symbols within your sources and their attributes
- `vECU` folder:
 - `buildArtifacts` folder: Library file and its associated debug information
 - `CMake` folder: CMake project artifacts
 - `imported` folder: all imported artifacts
 - `CMakeLists.txt`: set of directives and instructions for building your sources
- `1_Import.bat/1_Import.sh` file to trigger the [Importing Files and Folders](#) stage.

- `2_Build.bat/2_Build.sh`
file to trigger the [Building the vECU](#) stage.
- `3a_CheckFMU.bat/3a_CheckFMU.sh`
file to invoke FMPy and inspect the vECU outputs.
- `3b_StartDebugger.bat/3b_StartDebugger.sh`
file to invoke MSVC or VS Code as debugger.
- `3c_ShowSymbolDetails.bat/3c_ShowSymbolDetails.sh`
file to invoke Notepad++ (Windows) / new Terminal (Ubuntu 20.04 LTS)
and display the Symbol Details.
- `3d_RemoveGoLicense.bat/3d_RemoveGoLicense.sh`
file to remove the GO license from the vECU (only relevant if vECU was built
with GO-license).
- `SimpleExample.fmu`
release version of your vECU, for more details see [Simple Example](#).
- `SimpleExample_debug.fmu`
debug version of your vECU, for more details see [Simple Example](#).
- `vEcuConf.yaml`
the YAML configuration file, for more details see [Configuration](#).

4.6 Configuration

The YAML file contains the configurations for the import and build process as well as for the vECU itself. It is the only configuration you need to create and maintain. The YAML file is divided into several sections, each section configuring a particular attribute. You are guided through the YAML file with comments on each section and configuration attributes. Every section is structured in a standardized way:

```

1 #####
2 # The version of the .yaml file schema
3 #####
4 version:
5
6 #####
7
8 #####
9
10 #####
11
12 #####
13 #####
14 #####

```

A: comment with information on the corresponding section

B: configuration attributes and values

The following is a list of all attributes available in the YAML file:

- `version`

This is the version of the used YAML file schema and must not be changed.

- `build_mode`

You can select between 2 modes:

`build_sources`: You import source code (either as AUTOSAR Classic compliant or legacy C-code), header files, and static libraries. VECU-BUILDER then builds your vECU in the form of an FMU container.

The vECU will be named `<fmu_name>.fmu`.

`import_compiled`

You import an existing, already compiled and linked, software in the form of a DLL/SO containing the functionality of your vECU.

VECU-BUILDER then wraps it in an FMU container, sets up the inputs, outputs and tasks, patches the A2L file, sets up the XCP slave port, etc.

- `fmu_name`

Enter the name of your vECU.

The code of your vECU is located inside the FMU in the folder"

`resources/<fmu_name>.dll`".

This and other DLL/SO files are loaded and executed by the FMU runner.

– `import_into_project`

Enter the paths to the files and folders to be imported.

You can specify paths to folders and/or individual files such as *.c, *.h, *.cpp, *.hpp or *.zip archives which will be extracted during import.

The import target is the "vEcu/imported" folder in your workspace.

Environment variables can be used like this:

```
'${VECUBUILDER_EXAMPLES}\SimpleExample\src'
```

– `additional_resources`

Additional resources can be used to resolve dependencies by making .dll/.so libraries your application depends on part of the build and execution process. It is possible to reference files and folders, that the vECU assembly needs to run. A folder is copied recursively (not just the content of the folder) to the root of the resources folder. A file is copied to the root of the resources folder. any number of additional resources can be added.



Note

`additional_resources` does not support wildcards.

Specify all additional resources that are to be included in the FMU and they will be copied to the resources folder of the FMU during the Building FMU stage i.e.

```
${VECUBUILDER_WORKSPACE}/vECU/imported/additional_
DLLs/UsedByVECU.dll for Windows
```

Or

```
${VECUBUILDER_WORKSPACE}/vECU/imported/additional_
DLLs/UsedByVECU.so for Ubuntu 20.04 LTS.
```

It is possible to use plugins. You can include plugins as additional resources the same way. For more details, see [Template For Plugin V1 \(FM12\)](#).

– `import_external_compiled_vecu`

Only needed if you selected `import_compiled` as `build_mode`.

That DLL/SO already contains the code of your vECU, you can skip the compiling and linking and just import your DLL/SO into the FMU wrapper. Here you enter the DLL name and the path for updates:

`dll_so_name`: The name of the DLL/SO. There must exist a corresponding pdb file with the same filename.

`get_updates_from`: If VECU-BUILDER can find a DLL/SO and the pdb file in this folder, it will update the imported DLL/SO.

Environment variables can be used like this: `'${SystemDrive}/Sandbox'`.

– `architecture`

Specify the architecture.

When importing sources, the setting of this attribute has to match the integration and simulation system where the vECU is to be used.

In case you are importing an DLL/SO precompiled for either 32bit or 64bit architecture, this attribute must be set to the same.

 **Note**

For Ubuntu 20.04 only 64bit is supported.

– `xcp_slave`

Enter the port and IP address of the XCP Slave to be setup in your vECU.

These values are transferred to the patched A2L file. The used protocol is TCP. For more details, see [A2L File Patching](#).

 **Note**

- A socket (IP address + port + protocol) for the XCP connection between INCA and XCP slave can only be used once. If a port is busy, you must define another port in the YAML file.
- `xcp_slave` is supported for Windows only.

– `operating_system`

Enter the operating system. Currently only Windows and Ubuntu 20.04 LTS supported.

– `build_tool`

Enter your preferred build tool. `Build_tool` differs between Windows and Ubuntu 20.04 LTS.

Windows:

Set up the build tool to be used for your build.

 **Note**

VECU-BUILDER configures the build tool for the underlying CMake.

In case Visual Studio is selected, a Visual Studio Solution is generated.

If you choose MinGW Makefiles, a CMake project is generated.

These artefacts are stored in the “`vECU/CMake`” folder in your workspace.

- `path_to_mingw`: If the user-specific MinGW is defined, CMake builds the sources using this MinGW version.

Ubuntu 20.04 LTS:

You can choose Unix Makefiles.

— `cmake_generator_toolset`

Define which toolset should be used by CMake during the build process.

For more details, see [CMAKE_GENERATOR_TOOLSET](#).

— `inputs, outputs, parameters, locals`

Enter the variables you wish to expose as ports of your FMU.

Inputs, outputs, parameters, and locals refer to the causality of the FMI.

Wildcards of `*` and `?` are allowed. Arrays can be added using `myArray*`, the same goes for structures. If your wildcard expression breaks the YAML compatibility, put it in single apostrophes.

Example: `'*a'` finds all symbols ending with an 'a'.

Aliases can be defined for variables, which results in renaming of FMI ports.

The aliases are used in the `modelDescription.xml` and the original variable names are used in the `resources.txt`.

 **Note**

Variables of type enumeration will be interpreted as integers in the `modelDescription.xml` file of the FMU. The name-value mapping of enumerations will be ignored when enumerations are used as interfaces. Only the integer value will be exchanged.

— `initial_data`

Enter the path for source and target destination to define the initial values of calibration variables.

The initial data is virtually flashed into memory during initialization. The data file in the FMU (defined by destination) is read and its values are written to RAM. This simulates a part of the NVRAM (non-volatile RAM).

`source`: Where to get the file. During build-time this file will be copied from source.

`destination`: Where to store the file inside the FMU, relative to the resources folder of the FMU (optional). This file is used during run-time.

Supported formats:

`varVal`: list of pairs separated by one space, where the lhs refers to the C variable and the rhs to the value.

`dcm`: format containing ASAP2 labels and their values in physical form which are processed according to information in `a2l` file.

For more details, see [InitialData Functionality](#).

— `eeeprom`

Specify the eeeprom simulation attributes.

`source`: Path where to get the file. This is used during the build.

destination: Path where to store the file relative to the resources folder of the FMU. This is the working copy (optional).

sync: This can be a UNC Path or a regular pathname. When the vECU is initializing, this file is copied to the 'destination', if it exists. When the vECU terminates, the updated file in 'destination' is copied to the 'sync' location (optional). To setup the UNC Path, see [Windows Cannot Access Localhost While Using Sync Attribute in EEPROM](#).

c_variables: The C variable names that store the eeprom data.

Supported format:

.txt: A line starting with '#' is a comment. All other lines store the data stream to be flashed to the C variables. The order of the data stream lines is the same as the order of the c_variables listed.

A data stream is a sequence of bytes in hex format. Each byte is separated by a space. E.g.: 01 02 ee 4f. In the default YAML file the sync is commented out.

To get more information about eeprom, see [eeprom Functionality](#).

— tasks



Note

Task functions must have no arguments.

Define the tasks that are to be executed and their attributes. To simulate the microcontroller behavior with its periodically executed functions of your software, these functions are to be defined as tasks in this section. A function can be defined as a task only once, duplicated functions will be ignored.

function_name: '<function name>', without brackets, set in apostrophes, no arguments allowed.

trigger: Choose between cyclic, initial or terminate, the default is cyclic.

initial: Functions are called from `fmi2DoStep` before cyclic tasks .

cyclic: Functions are called from `fmi2DoStep` before terminate tasks .

terminate: Functions are called from `fmi2Terminate`.

fmi2_enter_init: Functions are called from `fmi2EnterInitializationMode`.

fmi2_exit_init: Functions are called from `fmi2ExitInitializationMode`.

period: <number> [in seconds], the default is 1.0.

first_call: <number> [in seconds] for the cyclic tasks, the default is period.

priority: The lower the number the higher the priority, the default is 0.

 **Note**

If two functions run at the same time, the one with the lower priority runs first.

`max_calls`: <number>, -1 means infinite, 0 means no call.

`trigger_function`: The function is written in `Multiply.c`. `trigger_function` only can be used if 'trigger' is 'event'. The trigger function predicts when the next event might occur and returns if the next events needs to be triggered. You can find the defined arguments of the trigger function in `Multiply.c`

`trigger_inputs`: A list of additional inputs that refer to variables accessible via symbol details. Trigger inputs must be included in the symbol details text file.

For more information, see [EventTrigger Example](#).

— [redirect_function_calls](#)

Enter the names functions to be replaced and their substitutes.

The function signatures of the two functions must be identical. This allows you to test the behavior of your software using alternative implementation without changing the original source code or to replace unfinished or hardware-dependent functions with mock functions.

`replaced_function`: Enter the function name of the function to be replaced.

`substitute_function`: The function name of the function that substitutes the replaced function.

 **Note**

Sometimes `redirect_function_calls` does not work as expected. For more details, see [additional_compile_flags](#) in this chapter and [Redirecting Function Calls Did Not Work as Expected](#).

— Only usable if you selected `build_sources` as `build_mode`.

You can select files and/or folders that should be included or excluded in/from the vECU build process.

Files are only included into the build if they are matched by at least one `build_include_filter` and are not matched by any `build_exclude_filter`.

— [assembly_list_files](#)

Specify your assembly list files for the build process.

Of the given sources defined by "`build_include_filters`" and "`build_exclude_filters`", only those listed in a file are passed to the compiler.

If no assembly list files are configured, all sources are compiled.

The same prerequisites as for "[additional_compile_flags](#)" on the previous page must be met. `additional_cxx_compile_flags` will be applied only to C++.

A successful use of `redirect_function_calls` depends on `additional_cxx_compile_flags`. Only if `additional_cxx_compile_flags` is set correctly, `redirect_function_calls` will work.

To prevent the GNU compiler from using incompatible optimizations when `redirect_function_calls` feature is enabled, optimizations are disabled by using the following flags:

```
# - '-fpermissive'
```

— [additional_static_libraries](#)

Only usable if you selected `build_sources` for `build_mode`.

The libraries need to be located in the folder `./projects/vEcu/imported`.

— [environment_variables](#)

You can define process-level environment variables that are set by the build process and by the FMI wrapper during the vECU execution.

Example: `PATH=c:/Temp; ${PATH}`

These variables can be configured and modified in one location and can be accessed from scripts and configuration files. Process-level environment variable of `VECUBUILDER_WORKSPACE` is created automatically during the build process with its value pointing to the current workspace.

— [additional_scripts](#)

Define your additional scripts for execution.

Project-specific scripts can be configured to be executed at various phases of the import and/or the build process.

You can utilize these to copy or modify files, add files to the FMU archive, parse files, etc. You may use Python, Perl, `.cmd` scripts, `.bat` batch/shell script files as long as these can be executed on your machine.

command: The script to be executed by the OS, the default search path is `${VECUBUILDER_WORKSPACE}/build/additional_scripts/` (utf-8 only).

trigger: Select when should your script be executed from these options:

- `before_import`
- `after_import`
- `before_build_sources`
- `before_build_fmus`
- `after_build_fmus`

priority: Define with which priority should your script be executed. The lower the number the higher the priority. Default value is 1.

For more details, see [Example of Additional Scripts: A2L Characteristics as Parameters](#).

– `patch_a2l_file`

An A2L file is required to connect an MCD tool such as INCA to the running vECU. The A2L file needs to be located in the folder: "vEcu/imported".

– `filename`: Enter the name of your A2L file to be patched.

– `symbol_name_mapping`

When using A2L or dcm files, ASAP2 labels might differ from the symbol names. If both, DCM and A2L, files are provided, then for each DCM entry, an A2L entry of the identical ASAP2 label name must exist.

based_on_csv:

csv file, where lhs is the symbol name and rhs is the ASAP2 label. The csv file must follow the following format: `SYMBOL_name;ASAP2_label`. You need to use a semicolon as a delimiter.

When using a csv file, you only can use simple string search & replace. Make sure that there are no header or any comments in the csv file and every line is treated as a data record.



Note

VECU-BUILDER removes all leading and trailing spaces before and after the first character.

Example:

```
my_symbol ; ASAP2_LABEL_5
```

This is a valid entry and will be treated as defined below:

```
my_symbol;ASAP2_LABEL_5
```

based_on_adx:

If mapping between ASAP2 labels and symbols is available in an adx file format (proprietary format of Bosch, only used within Bosch projects), this file can be made part of the build in order to apply the mappings. For mappings in adx file format, simple string search & replace is applied.



Note

The content of an adx file will be processed as is, no interpretation or validation will be performed by VECU-BUILDER. The file is assumed to be complete and correct.

based_on_assignments:

If the right-hand side includes a dollarsign '\$' (like in a reference to a group, e.g. (\$1), then a regular expression search & substitute is applied, else a simple string search and replace is applied.

One such regular expression allows to map multiple names at once. To see an example, see the following table.

| | | | |
|---------|------------------|----|---------|
| RegEx | (array)\[(\d+)\] | -> | \$1_\$2 |
| Mapping | array[1] | -> | array_1 |

The mappings can be verified by examining the json files appended to the debug FMU, located in `resources\mappings` folder.

VECU-BUILDER will update the memory addresses of entries in the provided A2L file. The original A2L file is renamed by appending `.bak` to its name. For more details, see [A2L File Patching](#) and [A2L Name Mapping](#).

– `debug_hook`

Specify whether to enable or disable a debug hook. When enabled, the FMU execution is interrupted when the FMU is instantiated until a debugger is attached. For more details, see [Debugging vECU](#).

– `additional_link_flags`

Only usable if you selected `build_sources` as `build_mode`.

Specify how the linker should work. Each individual flag must be written in a separate line and put in single apostrophes, i.e. `'/DEBUG'`.

The flags are written into the `CMakeLists.txt`.

For more details, see [MSVC linker options](#) or [gcc linker options](#).

– `simple_file_modifications`

Specify file modifications that shall be applied to files imported in `"vECU/imported"` folder.

In case you specify multiple modifications, they will be applied sequentially following the order in which they were specified.

The next two attributes are mandatory for all types of modifications.

`file_regex`: Specify the search RegEx for a file or a set of files that shall be modified.

`trigger`: Specify when the modification shall be applied from the 2 below options:

- `after_import` (default)
- `before_build_sources`

You can specify a single or multiple actions (modification types) from the 4 below options:

- `comment_line`: Comment out a single line of code by adding `'//'` at the beginning of the line.
- `search_and_replace`: Replace a line of code that matches the

Search_regex with the replacement.

- `insert_code_above`: Insert code above a matched line.
- `insert_code_below`: Insert code below a matched line.

You must specify `line_regex` and to which match(es) the modification are to be applied to (`apply_to`) for each action from the below 3 options:

- `all_matches` (default)
- `last_match`
- `first_match`

For `insert_code_above` and `insert_code_below`, you must specify the code section that is to be inserted.

When using `simple_file_modifications`, consider the following procedure to make sure, modifications are not included in `.bak` file.

1. Get the set of files and apply the file filter.
2. Revert backups for all files to be modified: Move the `.bak` files to overwrite the normal filename.
→ The backup file is deleted.
3. Create the backup on all files that need to be modified, excluding files ending with `.bak`.
4. Apply the file modifications to all files that need to be modified.

Note

If you need more sophisticated file modifications, use a project-specific script via the `additional_scripts`.

— `include_symbol_details`

The use of plugins may require that the release FMU contains symbol information. For example, if you want to change the cycle time of `"task_10ms"` using a plugin, then the symbol name `"task_10ms"` must be disclosed in the release vECU.

For more details about release FMU, see [Difference Between Debug and Release vECUs](#) and [Keeping Symbol Information in a Release FMU](#).

`fmi_<type>`: disabled (default) or enabled. When enabled, all symbol details of that fmi type will be included.

`symbol_names`: All symbol details matching the regular expressions will be included.

— `fmi`

You can use the functional mockup interface. Default version is 2.0. The use of version 3.0 is also possible. Use the following convention:

```
#fmi: '2.0'
```

More information about FMI can be found [here](#).

5 Exploring the Examples/Templates

This chapter contains details on examples/templates that are designed for users to get to know the features of VECU-BUILDER.

5.1 Simple Example

If you followed the instructions in the chapter Working with VECU-BUILDER, you now have a workspace on your computer which is based on the Simple Example.

5.1.1 FMPy

To conduct a quick smoke test of the created vECU, FMP is delivered along with VECU-BUILDER. This tool can be invoked via the `3a_CheckFMU.bat` on **Windows** or `3a_CheckFMU.sh` on **Ubuntu 20.04 LTS**. Simply execute this file to run the release vECU or drag-and-drop the debug vECU into this batch/shell script file to run the debug vECU.

FMPy opens a terminal where details of the FMU are displayed. Furthermore a graph showing the time and values of defined outputs is displayed in a browser window.



```

C:\Windows\system32\cmd.exe
[FMI] fmi2DoStep(component=0x1815c6de2f0, currentCommunicationPoint=9.94, communicationStepSize=0.01, noSetFMUStatePriorToCurrentPoint=1) -> OK
[FMI] fmi2GetReal(component=0x1815c6de2f0, vr=[3], nvr=1, value=[2.0]) -> OK
[FMI] fmi2DoStep(component=0x1815c6de2f0, currentCommunicationPoint=9.950000000000001, communicationStepSize=0.01, noSetFMUStatePriorToCurrentPoint=1) -> OK
[FMI] fmi2GetReal(component=0x1815c6de2f0, vr=[3], nvr=1, value=[2.0]) -> OK
[FMI] fmi2DoStep(component=0x1815c6de2f0, currentCommunicationPoint=9.96, communicationStepSize=0.01, noSetFMUStatePriorToCurrentPoint=1) -> OK
[FMI] fmi2GetReal(component=0x1815c6de2f0, vr=[3], nvr=1, value=[2.0]) -> OK
[FMI] fmi2DoStep(component=0x1815c6de2f0, currentCommunicationPoint=9.97, communicationStepSize=0.01, noSetFMUStatePriorToCurrentPoint=1) -> OK
[FMI] fmi2GetReal(component=0x1815c6de2f0, vr=[3], nvr=1, value=[2.0]) -> OK
[FMI] fmi2DoStep(component=0x1815c6de2f0, currentCommunicationPoint=9.98, communicationStepSize=0.01, noSetFMUStatePriorToCurrentPoint=1) -> OK
[FMI] fmi2GetReal(component=0x1815c6de2f0, vr=[3], nvr=1, value=[2.0]) -> OK
[FMI] fmi2DoStep(component=0x1815c6de2f0, currentCommunicationPoint=9.99, communicationStepSize=0.01, noSetFMUStatePriorToCurrentPoint=1) -> OK
[FMI] fmi2GetReal(component=0x1815c6de2f0, vr=[3], nvr=1, value=[2.0]) -> OK
[FMI] fmi2Terminate(component=0x1815c6de2f0) -> OK
[FMI] fmi2FreeInstance(component=0x1815c6de2f0)

```

Fig. 5-1: FMPy output

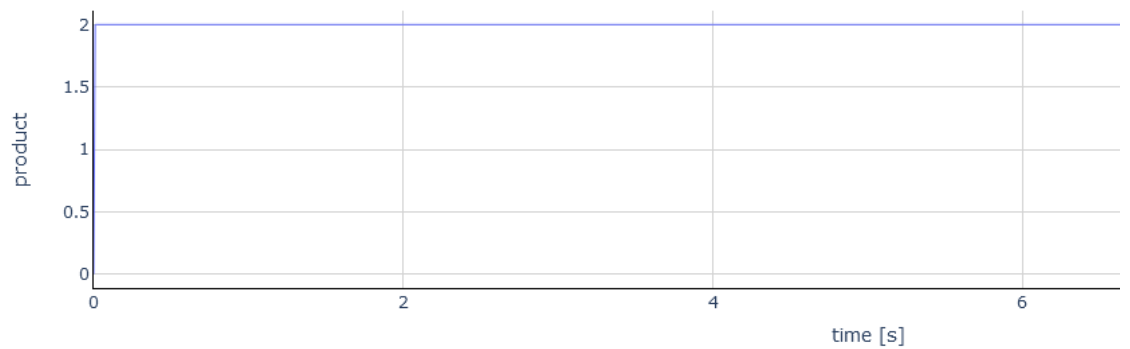


Fig. 5-2: Graph of defined outputs

An FMU, that is built by VECU-BUILDER will set the environment variable `VECUBUILDER_FMURESOURCES`. The environment variable is set for the process that runs the FMU. It is not set on system-level or user-level.

This environment variable stores the absolute path to the resources folder of the FMU. The environment variable can be found when inspecting the process properties in a process monitor tool.

5.1.2 Difference Between Debug and Release vECUs

You find two FMUs in this workspace, one named `SimpleExample.fmu` (which will be referred to as 'release vECU') and the other one named `SimpleExample_debug.fmu` (which will be referred to as 'debug vECU').

Extract each of these two FMU archives into its own folder and let's explore what they contain and how they differ.

The functional behavior of both vECUs is identical.

The debug vECU contains symbol information and additional artefacts, e.g., PDB (when build tool is MSVC) or DIE (when build tool is MinGW). Use the debug vECU to debug and step through your code.

When you compare the two extracted folders, you will notice that the main difference is in the resources folder.

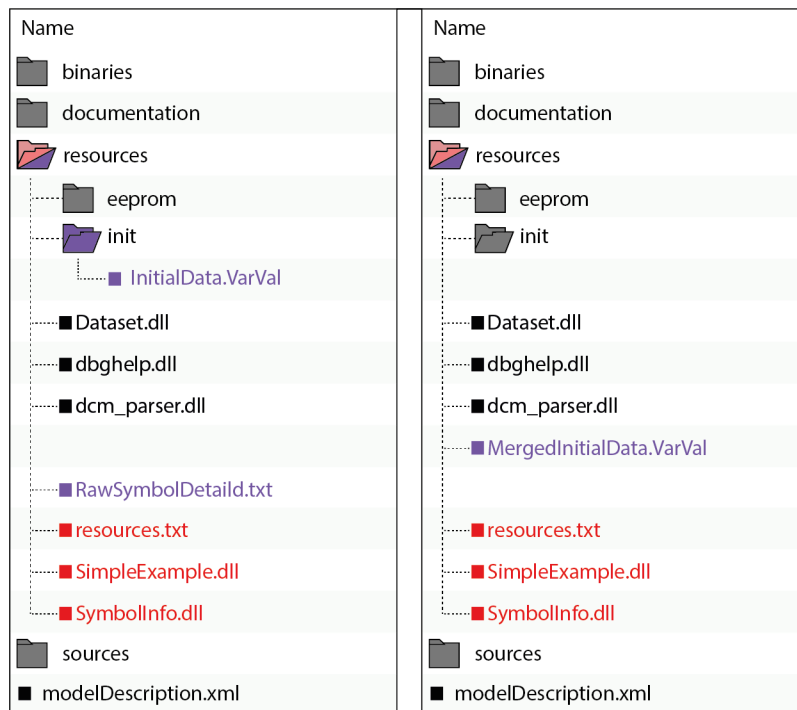


Fig. 5-3: Comparison of debug and release vECU (GCC compiler)

The release vECU contains only address information, unlike the debug vECU which contains the variables and function names. The release vECU protects the IP contained in the vECU and does not contain symbol information. Use the release vECU if you want to share it with others.

| vEcuDll SimpleExample.dll | vEcuDll SimpleExample.dll |
|-------------------------------|-------------------------------------|
| fmiVariables | fmiVariables |
| factor1 | 0x0000000000003028 8 10 0 0 |
| factor2 | 0x0000000000003028 8 10 0 0 |
| eeeprom_block_a.lifetime_ms | 0x0000000000008150 8 8 0 0 |
| eeeprom_block_a.poweron_count | 0x0000000000008150 2 4 0 0 |
| eeeprom_block_b.last_product | 0x0000000000008140 8 10 0 0 |
| product | 0x0000000000008060 8 10 0 0 |
| Tasks | Tasks |
| terminate 9 1.0 1.0 0 -1 | 0x0000000000001500 9 1.0 1.0 0 -1 |
| task_10 ms 2 0.01 0.01 2 -1 | 0x00000000000014c0 2 0.01 0.01 2 -1 |
| InitialData | InitialData |
| init/InitialData.VarVal | MergedInitialData.VarVal |
| Eeprom | Eeprom |
| eeeprom/1.txt | eeeprom/1.txt |
| eeeprom_block_a | 0x0000000000008150 22 |
| eeeprom_block_b | 0x0000000000008140 8 |

Fig. 5-4: Comparison of resources.txt

5.1.2.1 Keeping Symbol Information in a Release FMU

When using a plugin in your vECU, variables and functions are accessed by name. This is possible in a debug FMU, but not in a release FMU.

Searching for symbols in the plugin it is done by name. In the release FMU all symbols names are replaced by their memory addresses. When FMU Runner runs in release mode, it accesses the symbols directly by addresses and not by symbol names. As a reason, SymbolDetails file like `RawSymbolDetails.txt` file has to be kept in `release.fmu` file

Note that the release FMU does not contain the PDB/DWARF file. Thus, you can not debug the release FMU. `RawSymbolDetails.txt` is a subset of `SymbolDetails.txt` and is used by the plugin.

For more details, see [include_symbol_details](#) in Configuration chapter.

For more details about the Plugin Feature, see [Template For Plugin V1 \(FM12\)](#)

5.1.3 InitialData Functionality

Typically, software function and its data are separated. While the logic of the software function is defined in the source files, the data is stored in separate files in various formats. Common formats for such calibration data are DCM and CDF.

VECU-BUILDER provides support of DCM format, more details can be found in [DCM File Format](#). A `.dcm` file stores the data in their physical form that typically need to be processed into ECU-internal form. This processing is done based on `COMPU_METHOD` and `RECORD_LAYOUT` entries in an `a2l` file which the user needs to provide in the `patch_a2l_file` attribute of the `yaml` file.

In case the Symbols do not match the ASAP2 labels (entries in the `.dcm` and `.a2l` files), you can resolve this by applying mappings. These mappings are then used to map ASAP2 labels to their respective symbols and can be defined in one of these three ways:

- direct definition in the yaml file making use of regular expressions
- via .adx file
- via .csv file

More details on these options can be found under [symbol_name_mapping](#) section in Configuration chapter.

You also can define the initial data in the VarVal format. These initial data will not be processed based on entries in the a2l file neither will any mapping be applied. Thus the VarVal file shall contain the symbols and the ECU-internal values.

Simple Example contains sample files of both supported formats which can be found in folder

C:/ProgramData/ETAS\VECU-BUILDER/Examples_1.6.0/SimpleExample/init for **Windows**

Or

/opt/etas/VECU-BUILDER/Examples_1.6.0 for **Ubuntu 20.04**.

The .yaml file is preconfigured to make use of the InitialData.VarVal.

To experiment with .VarVal functionality

1. Open `Multiply.c` file located within your workspace in folder `vECU/imported`.

In `Multiply.c` the variables are defined. Variables **factor1** and **factor2** are the two inputs with the assigned values of 1 and 2. Variable **product** is the output and is calculated as the product of **factor1** and **factor2**.

As `InitialData.VarVal` file is already activated in the .yaml file, it is already used in the default Simple Example vECU.

2. Close the source file and navigate back to the workspace.
3. You can check the output using

`3a_CheckFMU.bat` on **Windows**

Or

`3a_CheckFMU.sh` on **Ubuntu 20.04 LTS**, described in [FMPy](#).

- ⇒ The output for SimpleExample is 2.
4. Change the value for factor 1 to 4.

5. Save the change.
6. Rebuild the the vECU using

2_Build.bat in the workspace on **Windows**

Or

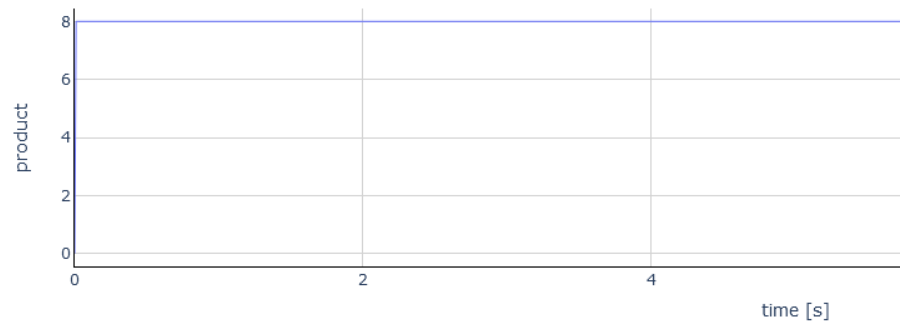
2_Build.sh in the workspace on **Ubuntu 20.04 LTS**.

- To show the changed output in the FMU, execute 3a_CheckFMU.bat on **Windows**

Or

3a_CheckFMU.sh on **Ubuntu 20.04 LTS**.

⇒ The new output is now 8.



- ⇒ The initial data set in the VarVal file are thus correctly used in the vECU.
- ⇒ The sources in Multiply.c stay the same. The variables are overwritten at run-time by the values of the InitialData.VarVal.

To experiment with initialData.dcm

- Open the .InitialData.dcm in Examples/SimpleExample/init.
- Change the value for factor 1 to 4.
- Change the value for factor 2 to 4.
- Save the changes.

```
FESTWERT factor1
  LANGNAME ""
  EINHEIT_W ""
  WERT 4.0
END

FESTWERT factor2
  LANGNAME ""
  EINHEIT_W ""
  WERT 4.0
```

- Uncomment the source and destination for InitialData.VarVal.
- Comment the source and destination for InitialData.dcm.

```
initial_data:
#- source: '${VECUBUILDER_EXAMPLES}\SimpleExample\init\InitialData.VarVal'
# destination: 'init/InitialData.VarVal'
- source: '${VECUBUILDER_EXAMPLES}\SimpleExample\init\InitialData.dcm'
  destination: 'init/InitialData.dcm'
```

- Rebuild the the vECU using
2_Build.bat in the workspace on **Windows**

Or

2_Build.sh in the workspace on **Ubuntu 20.04 LTS**.

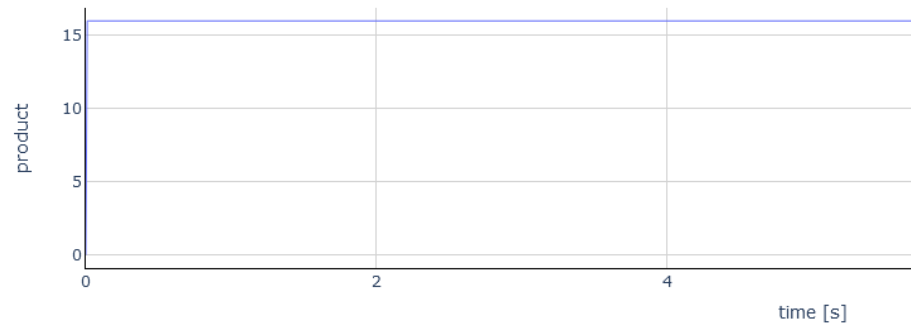
8. To show the changed output in the FMU, execute

3a_CheckFMU.bat on **Windows**

Or

3a_CheckFMU.sh on **Ubuntu 20.04 LTS**.

⇒ The new output is now 16.



⇒ The initial data set in the dcm file are thus correctly used in the vECU.

⇒ The sources in Multiply.c stay the same. The variables are overwritten at run-time by the values of the InitialData.dcm.



Note

Several files and formats can be defined. If one variable is set in multiple files, the value of the last file is used.













For release vECU all initial data is merged into MergedInitialData.VarVal. This VarVal file protects the IP. Release and debug vECU behave the same. To get the different folder structures, see [Difference Between Debug and Release vECUs](#).

5.1.4 eeprom Functionality

The eeprom data is loaded from a file to RAM during vECU initialization. The data is saved to the file before running terminate tasks and when unloading the vECU. This can be used to simulate a soft reset behavior where EEPROM stored data are preserved and not lost once the simulation of vECU terminates. A typical application of this feature is the storage of total mileage information in the ESP controller.













1. Open `vEcuConf.yaml` file of SimpleExample and navigate to eeprom section.

With standard configuration, `eeprom_data.txt` was copied from `VECUBUILDER_EXAMPLES/SimpleExample/src` into the workspace to `vECU/imported` during the import. During `SimpleExample.fmu` build, `eeprom_data.txt` file is integrated into the FMU as `1.txt` in `resources/eeprom` folder. This happens because optional **destination** attribute is active. You can change the destination path and file name accordingly. This file is the working copy.

| vECU > imported | SimpleExample.fmu > resources > eeprom | | | | | | |
|--|--|---|--|--|--|------|---|
| <table border="1"> <thead> <tr> <th>Name</th> </tr> </thead> <tbody> <tr> <td> eeprom_data.txt</td> </tr> <tr> <td> Multiply.c</td> </tr> <tr> <td> Multiply.h</td> </tr> </tbody> </table> | Name |  eeprom_data.txt |  Multiply.c |  Multiply.h | <table border="1"> <thead> <tr> <th>Name</th> </tr> </thead> <tbody> <tr> <td> 1.txt</td> </tr> </tbody> </table> | Name |  1.txt |
| Name | | | | | | | |
|  eeprom_data.txt | | | | | | | |
|  Multiply.c | | | | | | | |
|  Multiply.h | | | | | | | |
| Name | | | | | | | |
|  1.txt | | | | | | | |

Note

If destination attribute is deactivated, `eeprom_data.txt` is integrated into the FMU in `resources` folder.

| vECU > imported | SimpleExample.fmu > resources | | | | | | |
|--|-------------------------------|---|--|--|--|------|---|
| <table border="1"> <thead> <tr> <th>Name</th> </tr> </thead> <tbody> <tr> <td> eeprom_data.txt</td> </tr> <tr> <td> Multiply.c</td> </tr> <tr> <td> Multiply.h</td> </tr> </tbody> </table> | Name |  eeprom_data.txt |  Multiply.c |  Multiply.h | <table border="1"> <thead> <tr> <th>Name</th> </tr> </thead> <tbody> <tr> <td> eeprom_data.txt</td> </tr> </tbody> </table> | Name |  eeprom_data.txt |
| Name | | | | | | | |
|  eeprom_data.txt | | | | | | | |
|  Multiply.c | | | | | | | |
|  Multiply.h | | | | | | | |
| Name | | | | | | | |
|  eeprom_data.txt | | | | | | | |

2. Open `eeprom_data.txt` in `imported` folder and check the content.

Note

`eeprom_data.txt` must include the relevant data in expected HEX format.

3. Go back to yaml file.

In the standard configuration the following **c_variables** are used:

or

2_Build.sh in the workspace on **Ubuntu 20.04 LTS**.

8. To start the simulation, execute

3a_CheckFMU.bat on **Windows**

or

3a_CheckFMU.sh on **Ubuntu 20.04 LTS**.

9. Navigate to

C:/TEMP on **Windows**

or

//localhost/c\$/TEMP on **Ubuntu 20.04**.

⇒ eeprom_data.txt was added to sync location.

```
# This is a comment
# eeprom_block_a
20 4e 00 00 00 00 00 00 02 00
# eeprom_block_b
00 00 00 00 00 00 00 40
```

5.1.5 Features to Explore in the Simple Example Workspace

Now start experimenting with the following features in this current workspace:

- build tool, inputs, outputs and tasks
- Initial data and eeprom
- redirect function calls
- debug hook

5.2 BCU Example (Only Available for Windows)

To create a workspace based on the BCU example, follow the steps described in [Creating a New Workspace](#) to the point where the YAML file opens in Notepad++.

1. Replace the entire content of the YAML file with the content of prepared BCU configuration YAML file located in:

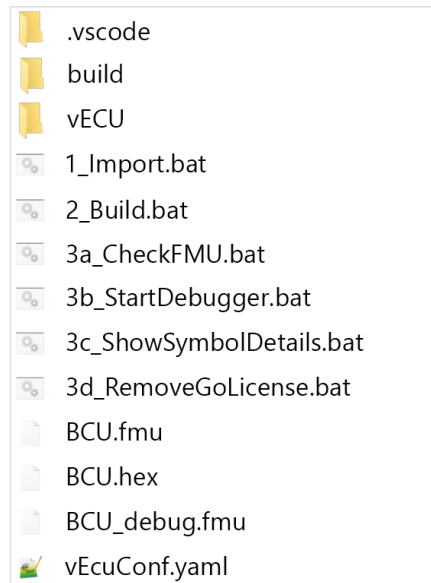
`C:/ProgramData/ETAS/VECU-BUILDER/Examples_1.6.0/BCU for Windows.`

In the `.yaml` file a2l file patching is enabled..

2. Continue the process as described in [Working with VECU-BUILDER](#).
- ⇒ With the enabled a2l file patching, hex file is generated during the build process and is available in the workspace after the build. For more information see [patch_a2l_file](#) and [Hex File Generation](#).

Note

The hex file will only be part of the workspace if the a2l file patching is activated.



5.2.1 Show Symbol Information

To see all the symbols available in your vECU, open the `SymbolDetails` file.

1. Run the command:

`3c_ShowSymbolDetails.bat` on **Windows**.

⇒ A text editor window (Windows) / a new terminal (Ubuntu 20.04 LTS) opens, and symbol details are shown.

```

1 217 symbol(s) found, 1 symbol(s) with non-unique name(s).
2 Address      ;Size      ;Type      ;BF-ofs;BF-Len;Value      ;Symbol Name
3 0x0000000000001330;0          ;Function  ;0      ;0      ;          ;_add_factor1_factor2_sum
4 0x0000000000001360;0          ;Function  ;0      ;0      ;          ;_multiply_factor1_factor2_product
5 0x0000000000001390;0          ;Function  ;0      ;0      ;          ;_task_10ms
6 0x00000000000013d0;0          ;Function  ;0      ;0      ;          ;_terminate
7 0x00000000000013e8;0          ;Function  ;0      ;0      ;          ;_dllmainCRT_dispatch
8 0x000000000000144c;0          ;Function  ;0      ;0      ;          ;_dllmainCRT_process_attach
9 0x00000000000015a8;0          ;Function  ;0      ;0      ;          ;_dllmainCRT_process_detach
10 0x0000000000001650;0          ;Function  ;0      ;0      ;          ;_dllmain_dispatch
11 0x00000000000017d0;0          ;Function  ;0      ;0      ;          ;_dllmain_raw
12 0x00000000000017f0;0          ;Function  ;0      ;0      ;          ;_CRT_INIT
13 0x00000000000017f8;0          ;Function  ;0      ;0      ;          ;_DllMainCRTStartup
14 0x0000000000001844;0          ;Function  ;0      ;0      ;          ;_get_entropy
15 0x00000000000018cc;0          ;Function  ;0      ;0      ;          ;_security_init_cookie
16 0x00000000000019a4;0          ;Function  ;0      ;0      ;          ;_DllMain
17 0x00000000000019d0;0          ;Function  ;0      ;0      ;          ;_srt_initialize_type_info
18 0x00000000000019e4;0          ;Function  ;0      ;0      ;          ;_srt_uninitialize_type_info
19 0x00000000000019f4;0          ;Function  ;0      ;0      ;          ;_local_stdio_printf_options
20 0x0000000000001a00;0          ;Function  ;0      ;0      ;          ;_local_stdio_scanf_options
21 0x0000000000001a0c;0          ;Function  ;0      ;0      ;          ;_srt_initialize_default_local_stdio_options
22 0x0000000000001a30;0          ;Function  ;0      ;0      ;          ;_srt_narrow_argv_policy::configure_argv

```

Fig. 5-5: Symbol Details of BCU example (Windows)

5.2.2 A2L File Patching

Most ECU software authoring tools can generate an A2L file for you. It contains the addresses of your labels for a specific target. In addition, it may contain tool-specific statements or even non-standard clauses. The label addresses of a vECU target differ from the addresses of a physical ECU target which means the original A2L file cannot be used for an XCP connection with a vECU target.

Since the generation of A2L files is an intricate task, VECU-BUILDER excludes this functionality completely. Instead, VECU-BUILDER reads, modifies, and writes a given A2L file. This patching procedure preserves most of the original contents of the A2L file but changes all addresses to those of the vECU target. A backup copy of the original A2L file is preserved (named as `*.a2l.bak`).



Note

The A2L patching leads to an A2L file that works in ETAS INCA. This file may not work in Vector CANoe or CANape.

VECU-BUILDER includes its own XCP slave software component. Currently, it supports TCP connections only. The communication parameters for an XCP connection are part of an A2L file. VECU-BUILDER patches in the values for TCP port and IP address, which were specified in the YAML file. For instance:

| Original A2L file | Patched A2L file |
|--|--|
| <pre> /begin XCP_ON_TCP_IP 0x0100 /* XCP on IP 1.0 */ <TCPPORT> /* Port */ /ADDRESS "<IPADDR>" /end XCP_ON_TCP_IP </pre> | <pre> /begin XCP_ON_TCP_IP */0x0100 /* XCP on IP 1.0 */ 12345 /* Port */ ADDRESS "127.0.0.1" /end XCP_ON_TCP_IP </pre> |

If your A2L file contains an "XCP_ON_UDP_IP" clause, then VECU-BUILDER rewrites it to an "XCP_ON_TCP_IP" clause. The integrated XCP slave supports a limited subset of the commands of the ASAM MCD-1 (XCP) standard version 1.0. It supports a limited subset of the clauses from ASAM MCD-2 (ASAP2 / A2L) standard version 1.7.1.

If your ECU software includes an XCP slave already, you may want to remove this software component from your vECU software stack.

5.2.3 A2L Name Mapping

By default, the A2L file contains the symbol names of characteristics and measurements. Sometimes the symbol names in the A2L file are renamed. Because the addresses in the A2L must refer to the original symbol names, one must map them.

| Original A2L file | Mapped and patched A2L file |
|--|--|
| <pre> /begin CHARACTERISTIC Hys- teresis_LightOffIntensity "unsigned integer 16bit" VALUE 0x00000000 RTAA2L_Internal_Scalar_ UnsignedWord 0 CompuMethods_STEP_100_ OFFSET_0 0 100 DISPLAY_IDENTIFIER Hys- teresis_LightOffIntensity /end CHARACTERISTIC </pre> | <pre> /begin CHARACTERISTIC Hys- tLiOfInt "unsigned integer 16bit" VALUE 00x00003016 RTAA2L_Internal_Scalar_ UnsignedWord 0 CompuMethods_STEP_100_ OFFSET_0 0 100 DISPLAY_IDENTIFIER Hys- teresis_LightOffIntensity /end CHARACTERISTIC </pre> |

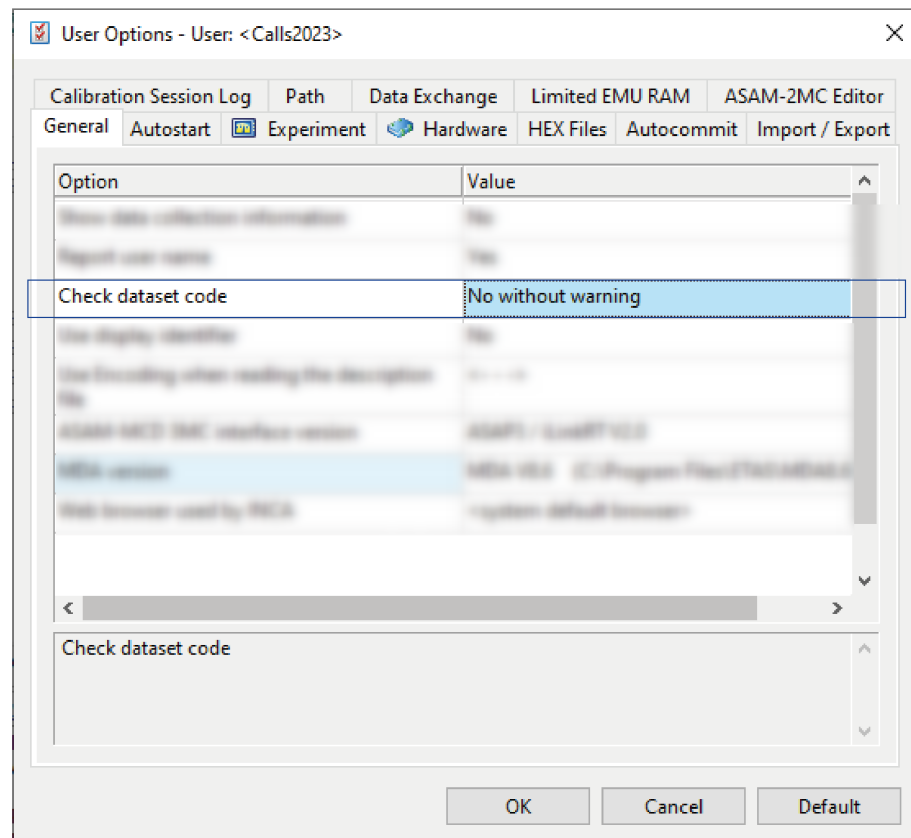
5.2.4 Hex File Generation

When the a2l patching mechanism is activated, VECU-BUILDER creates a hex file. The hex file (BCU.hex) is located in the corresponding workspace. You can use the hex file for working with ETAS INCA. The hex file contains the data INCA considers as the so-called reference page. INCA uses these data to calculate CRC check.

You can upload this generated hex file while creating an INCA experiment. After uploading the file, workflows within INCA will be enabled. For more information about INCA and hex file upload with INCA, see the corresponding INCA User Guides available in the [ETAS Download Center](#).

Use the following settings in INCA for a successful hex file upload:

1. Open **User Options**.
2. Click the **General** tab.
3. Go to **Check dataset code**.
4. Make sure that the value is **No without warning**.
5. Click **OK**.



Note

If the settings are not as described, INCA will display an error and the hex file will not be accepted.

5.2.5 Example of Additional Scripts: A2L Characteristics as Parameters

This is an example of how additional scripts can be used. With the following script, it is possible to use a2l characteristics as parameters of a vECU.

To run the script

1. Follow the steps described in [BCU Example \(Only Available for Windows\)](#) to create a BCU workspace.
2. Copy from `${VECUBUILDER_EXAMPLES}/BCU/additional_scripts`
 - `6_get_characteristics.bat`
 - `6_get_characteristics.py`

to `<My_BCU_Workspace>/build/additional_scripts`.

3. Open `<My_BCU_Workspace>/build/additional_scripts/6_get_characteristics.bat` script and adapt the path to the python interpreter according to your specific python installation.
4. In `vEcuConf.yaml` file uncomment the following lines in the `additional_scripts` section.

```
additional_scripts:
#-- command: .....1_before_import.bat
#... trigger: .....before_import
#... priority: .....5
#-- command: .....2_after_import.bat
#... trigger: .....after_import
#... priority: .....5
#-- command: .....3_before_build_sources.bat
#... trigger: .....before_build_sources
#... priority: .....5
#-- command: .....4_before_build_fmus.bat
#... trigger: .....before_build_fmus
#... priority: .....5
#-- command: .....5_after_build_fmus.bat
#... trigger: .....after_build_fmus
#... priority: .....5
- command: .....6_get_characteristics.bat
  trigger: .....after_import
  priority: .....5
```

5. Start building your workspace using `1_Import.bat` for **Windows**.
 - ⇒ Characteristics were added in `parameters` section of `vEcuConf.yaml` file and are also available as parameters in the built `BCU.fmu`.

```
parameters:
- Hysteresis_LightOffIntensity
- Hysteresis_LightOffTime
- Hysteresis_LightOnIntensity
- Hysteresis_LightOnTime
- Hysteresis_WiperOffIntensity
- Hysteresis_WiperOffTime
- Hysteresis_WiperOnIntensity
- Hysteresis_WiperOnTime
```


5.2.6 Features to Explore in the BCU Workspace

Now start experimenting with the following features in the current workspace:

- additional include directories
- additional compile and linker flags
- additional scripts
- xcp slave and A2L file patching and mapping
- Hex file generation
- Characteristics as inputs

5.3 EventTrigger Example

This example shows the possibility of using event-triggered tasks. Therefore a function and a trigger function are needed. The use of trigger inputs is optional. You can add up to 16 trigger inputs. This trigger function predicts when the next event might occur and returns if the next events needs to be triggered. You can find the event-triggered task in the `task` section in the `.yaml` file.

The function name is `teeth_count` and the trigger function is `tooth_event`.

The function and the trigger function are defined in the `Multiply.c` file in `VECU` folder of the corresponding workspace. Optional inputs of `trigger_function` must be included in `SymbolDetails.txt`.

```

1 19 symbol(s) found, 0 symbol(s) with non-unique name(s).
2 Address      ;Size      ;Type      ;BF-Ofs;BF-Len;Value      ;Symbol Name
3 0x0000000000001470;0          ;Function   ;0      ;0      ;          ;
   ;dummy_49d6c308_5c4d_4e35_ae01_3a51318a08c8
4 0x0000000000001480;0          ;Function   ;0      ;0      ;          ;multiply_factor1_factor2_product
5 0x00000000000014a7;0          ;Function   ;0      ;0      ;          ;add_factor1_factor2_sum
6 0x00000000000014ce;0          ;Function   ;0      ;0      ;          ;task_10ms
7 0x0000000000001529;0          ;Function   ;0      ;0      ;          ;tooth_event
8 0x0000000000001632;0          ;Function   ;0      ;0      ;          ;teeth_count
9 0x0000000000001648;0          ;Function   ;0      ;0      ;          ;terminate
10 0x0000000000003020;8          ;float64    ;0      ;0      ;1          ;factor1
11 0x0000000000003028;8          ;float64    ;0      ;0      ;2          ;factor2
12 0x0000000000003030;8          ;float64    ;0      ;0      ;0.00400000000000000001 ;reventTimeCheck
13 0x0000000000003038;1          ;u8         ;0      ;0      ;1          ;ignition
14 0x0000000000003040;8          ;float64    ;0      ;0      ;357.14285714285711      ;rpm
15 0x0000000000008060;8          ;float64    ;0      ;0      ;0          ;product
16 0x0000000000008068;4          ;s32        ;0      ;0      ;0          ;teeth_counter
17 0x0000000000008140;8          ;User-Defined Type;0      ;0      ;          ;reeprom_block_b
18 0x0000000000008140;8          ;float64    ;0      ;0      ;0          ;reeprom_block_b.last_product
19 0x0000000000008150;16         ;User-Defined Type;0      ;0      ;          ;reeprom_block_a
20 0x0000000000008150;8          ;u64        ;0      ;0      ;0          ;reeprom_block_a.lifetime_ms
21 0x0000000000008158;2          ;u16        ;0      ;0      ;0          ;reeprom_block_a.poweron_count
22
23 The following symbol names were non-unique (same name, various addresses).

```

Fig. 5-6: Symbol Details of EventTrigger Example(Windows)

It is checked, if the defined conditions of the trigger function are met. The default check time is 4ms. If the conditions are met, the function is called. E.g. if the conditions of trigger function `tooth_event` are met, the function `teeth_count` is called.

5.3.1 Event-Triggered Tasks

To create a workspace based on the EventTrigger example, follow the steps described in [Creating a New Workspace](#) to the point where the YAML file opens in Notepad++.

1. Replace the entire content of the YAML file with the content of prepared EventTrigger configuration YAML file located in:

`C:/ProgramData/ETAS/VECU-BUILDER/Examples_1.6.0/EventTriggerExample for Windows`

Or

/opt/etas/VECU-BUILDER/Examples_1.6.0/EventTriggerExample for **Ubuntu 20.04 LTS**.

2. Continue the process as described in [Working with VECU-BUILDER](#).
- ⇒ The workspace was created.

5.4 Template For Plugin V1 (FMI2)

5.4.1 Plugin Feature

With VECU-BUILDER plugin feature it is possible to implement an own logic at run-time, i.e. during the following phases of FMU Runner:

- Instantiation
- Initialization
- Step execution
- Task execution

The phases follow the rules of FMI2 standard. More information about FMI can be found [here](#).

The user-configured plugin implementation occurs in form of a CMake project. This CMake project needs to be used to implement the functionality of plugin interface. For more details, see [Plugin Interface](#).

Within `plugin_template_v1_FMI2` folder (see [Installed Files and Folders](#) for installation path), the following files and folders are installed:

- `include` folder: Contains the header files.
- `build.bat/build.sh`: Script, that contains a small list of commands which builds `plugin_template` shared object (DLL / SO).
- `CMakeLists.txt`: File, which contains CMake configuration for the `plugin_template` project.
- `plugin_template.cpp`: One of the two main 2 C++ files of the `plugin_template` project. These files include the `main()` function.
- `plugin_template.h`: One of the two main 2 C++ files of the `plugin_template` project.

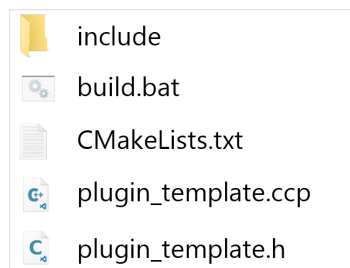


Fig. 5-7: Installed files and folders for plugin

Once you implemented your own logic, the plugin can be included in VECU-BUILDER as an additional resource.

The plugin project uses two header files that define VECU-BUILDER types. They are located in the `include` folder of the project template:

- `plugininterface.h`
- `vecubTypes.h`

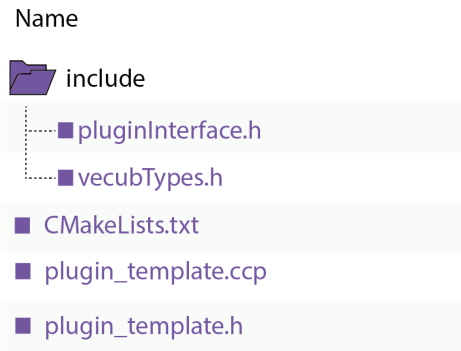


Fig. 5-8: plugin_template folder

As plugin feature is supported for FMI2 and FMI3, the functions for each plugin are grouped in `pluginInterface.h`.

For plugin version V1 (FMI2) look for `#define FMI2` and `#ifdef FMI2`.

When using plugins, the use of `include_symbol_details` is mandatory. For example, if you want to change the cycle time of "task_10ms" using a plugin, then the symbol name "task_10ms" must be disclosed in the release vECU.

5.4.2 Plugin Configuration

You can include plugins as additional resources:

- `additional_resources:`
 - `${VECUBUILDER_EXAMPLES}\plugin_template\CMake\Debug\plugin_template.dll` for **Windows**
- or*
- `additional_resources:`
 - `${VECUBUILDER_EXAMPLES}/Linux/plugin_template/CMake/libplugin_template.so` for **Ubuntu 20.04**.

Note

A plugin can be renamed. However it is mandatory to follow the conventions mentioned below in any case:

- A `*.dll` file (for Windows) must start with **plugin**. It then will be managed as plugin by VECU-BUILDER.
- A `*.so` file (for Linux) must start with **libplugin**. It then will be managed as plugin by VECU-BUILDER.

If the plugin does not follow this naming convention, VECU-BUILDER will not consider the `*.dll` or `*.so` as plugin.

The files are saved in `.fmu` file in `resources` folder. From here VECU-BUILDER will load them as plugins.

5.4.3 Plugin Interface

To see and use the plugin interface, open `pluginInterface.h` in `include` folder.

Plugin Functions

You can use several plugin functions. There are optional and mandatory plugin functions and therefore not all functions from plugin interface have to be implemented.

Pointer and Function Pointers

All plugin interface functions have one argument as a pointer of type `VecubCallbacks1`. The following Plugin Callback Functions are available. The trigger for each Plugin function call is related to the FMI protocol.

| Callback Function Name | Trigger | Priority |
|----------------------------------|---|-----------|
| <code>vecubPluginVersion</code> | During "fmi2Instantiate" and before "vecubInstantiate1" | mandatory |
| <code>vecubInstantiate1</code> | During "fmi2Instantiate" and after "vecubPluginVersion" | mandatory |
| <code>vecubFmi2EnterInit1</code> | During "fmi2EnterInitializationMode" | optional |
| <code>vecubFmi2ExitInit1</code> | During "fmi2ExitInitializationMode" | optional |
| <code>vecubPreDoStep1</code> | At the beginning of "fmi2DoStep" | optional |
| <code>vecubPostDoStep1</code> | At the end of "fmi2DoStep" | optional |
| <code>vecubPreTask1</code> | Before calling a task | optional |
| <code>vecubPostTask1</code> | After calling a task | optional |
| <code>Terminate1</code> | During "fmi2Terminate" | mandatory |

This `VecubCallbacks1` pointer keeps a list of function pointers which could be used in order for plugin to grab information from VECU-BUILDER.

Available functionalities (callbacks) are:

- logging
- get symbol info for global variables, tasks (address, type)
- get task object
- get symbol accessor (reading and writing global variables)

```

typedef void          (__cdecl* VecubCallbackLog1)          (vecubString);
typedef void*        (__cdecl* VecubCallbackGetSymbolInfo1) (vecubString); // returns SymbolInfo*
typedef int          (__cdecl* VecubCallbackGetTask1)      (vecubString, Task*&);
typedef PtrSymbolAccess& (__cdecl* VecubCallbackGetSymbolAccessor1) (vecubString);

using VecubCallbacks1 = struct VecubCallbacks1 {
    VecubCallbackLog1          log1;
    VecubCallbackGetSymbolInfo1  getSymbolInfo1;
    VecubCallbackGetTask1      getTask1;
    VecubCallbackGetSymbolAccessor1  getSymbolAccessor1;
};

```

Fig. 5-9: function pointers

Example for Usage of Callbacks

The following code visualizes how callbacks can be used for Plugin Version 1.

```

// ===== plugin vers. 01 =====
DllExport vecub1Status __cdecl vecubFmi2EnterInit1(const VecubCall-
backs1* vecubCallbacks)
{
    if (!vecubCallbacks)
        return vecub1Status::vecubError;
}

```

The code defines the function `vecubFmi2EnterInit1` that takes a pointer to the structure `VecubCallbacks1` as input. It checks if the passed pointer `vecubCallbacks` is valid. If not, it returns `vecub1Status::vecubError`.

```

// logging
if (vecubCallbacks->log1)
{
    vecubCallbacks->log1("user message");
}

```

It checks if the `log1` function is defined in `vecubCallbacks` structure. If yes, it calls the function with the argument `user message`.

```

// get symbol info
SymbolInfo* symbol = static_cast<SymbolInfo*>(vecubCallbacks-
>getSymbolInfo1("factor1"));
std::cout << "The symbol has address: " << std::hex << symbol-
>addr << std::endl;

```

It calls `getSymbolInfo1` function to obtain information about a symbol named `factor1`. The returned symbol is stored in the variable `symbol`. The address of the symbol is printed using `std::cout`.

```

// get symbol accessor
PtrSymbolAccess& symbolAccessor = vecubCallbacks->getSym-
bolAccessor1("factor1");
IntFloat64 val = (*symbolAccessor).get();
val.f = 5;
(*symbolAccessor).set(val);

```

It calls `getSymbolAccessor1` function to obtain an accessor for the `factor1` symbol. The accessor is stored in the reference variable `symbolAccessor`. It reads a value of type `IntFloat64` from the symbol and assigns the value 5 to it. Then, it writes the value back to the symbol.

```

// get task object
Task* task{nullptr};
vecubCallbacks->getTask1("task_10ms", task);
task->setTNextCall(123);
if (task) {
    std::cout << "FOUND! task: " << " ~ " << task->getName() <<
std::endl;
}

```

It calls `getTask1` function to obtain a task object with the name `task_0ms`. The task object is stored in the pointer variable `task`. `setTNextCall` function is called on the task object with the value `123` as the argument. If the task object is valid, a message is printed with the task's name.

```

return vecub1Status::vecubOK;
}

```

It returns `vecub1Status::vecubOK`.

5.4.4 What a Plugin Can Do With Tasks

A plugin can access a task defined in VECU-BUILDER and can change different properties of a task at run-time.

As described in [Plugin Functions](#), you can use several functions. One of these functions is used to get a task. The used function pointer is `getTask1`, which returns a task type. For more details, see [Fig. 5-9](#)

Once the plugin access a task, it can manage through the task interface and the behavior of a task during run-time.

The interface of the task is available in Plugin template project through the definition of `task` class in `include/vecubTypes.h` file.

In this class, you can see specific functions of task class which manage i.e the task name, task period, or task priority.

5.5 Template For Plugin V2 (FMI3)

5.5.1 Plugin Feature V2

With VECU-BUILDER plugin feature it is possible to implement an own logic at run-time, i.e. during the following phases of FMU Runner:

- Instantiation
- Initialization
- Step execution
- Task execution

The phases follow the rules of FMI3 standard. More information about FMI can be found [here](#).

The user-configured plugin implementation occurs in form of a CMake project. This CMake project needs to be used to implement the functionality of plugin interface. For more details, see [Plugin Interface](#).

Within `plugin_template_v2_FMI3` folder (see [Installed Files and Folders](#) for installation path), the following files and folders are installed:

- `include` folder: Contains the header files.
- `build.bat/build.sh`: Script, that contains a small list of commands which builds `plugin_template` shared object (DLL / SO).
- `CMakeLists.txt`: File, which contains CMake configuration for the `plugin_template` project.
- `plugin_template.cpp`: One of the two main 2 C++ files of the `plugin_template` project. These files include the `main()` function.
- `plugin_template.h`: One of the two main 2 C++ files of the `plugin_template` project.

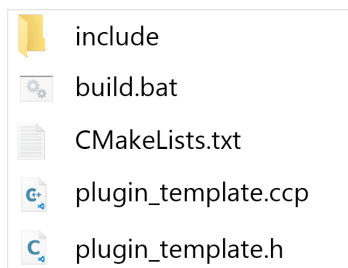


Fig. 5-10: Installed files and folders for plugin

Once you implemented your own logic, the plugin can be included in VECU-BUILDER as an additional resource.

The plugin project uses two header files that define VECU-BUILDER types. They are located in the `include` folder of the project template:

- `plugininterface.h`
- `vecubTypes.h`

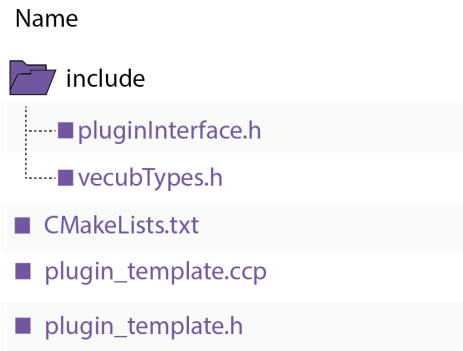


Fig. 5-11: plugin_template folder

As plugin feature is supported for FMI2 and FMI3, the functions for each plugin are grouped in `pluginInterface.h`.

For plugin version V2 (FMI3) look for `#define FMI3` and `#ifdef FMI3`.

When using plugins, the use of `include_symbol_details` is mandatory. For example, if you want to change the cycle time of `"task_10ms"` using a plugin, then the symbol name `"task_10ms"` must be disclosed in the release `VECU`.

5.5.2 Plugin Configuration V2

You can include plugins as additional resources:

- `additional_resources:`
 - `${VECUBUILDER_EXAMPLES}\plugin_template\CMake\Debug\plugin_template.dll` for **Windows**
- or*
- `additional_resources:`
 - `${VECUBUILDER_EXAMPLES}/Linux/plugin_template/CMake/libplugin_template.so` for **Ubuntu 20.04**.



Note

A plugin can be renamed. However it is mandatory to follow the conventions mentioned below in any case:

- A `*.dll` file (for Windows) must start with **plugin**. It then will be managed as plugin by `VECU-BUILDER`.
- A `*.so` file (for Linux) must start with **libplugin**. It then will be managed as plugin by `VECU-BUILDER`.

If the plugin does not follow this naming convention, `VECU-BUILDER` will not consider the `*.dll` or `*.so` as plugin.

The files are saved in `.fmu` file in `resources` folder. From here `VECU-BUILDER` will load them as plugins.

5.5.3 Plugin Interface V2

To see and use the plugin interface, open `pluginInterface.h` in `include` folder.

Plugin Functions

You can use several plugin functions. There are optional and mandatory plugin functions and therefore not all functions from plugin interface have to be implemented.

Pointer and Function Pointers

All plugin interface functions have one argument as a pointer of type `VecubCallbacks2`. The following Plugin Callback Functions are available. The trigger for each Plugin function call is related to the FMI protocol.

| Callback | Trigger | Priority |
|----------------------------------|---|-----------|
| <code>vecubPluginVersion</code> | During "fmi3Instantiate" and before "vecubInstantiate2" | mandatory |
| <code>vecubInstantiate2</code> | During "fmi3Instantiate" and after "vecubPluginVersion" | mandatory |
| <code>vecubFmi3EnterInit2</code> | During "fmi3EnterInitializationMode" | optional |
| <code>vecubFmi3ExitInit2</code> | During "fmi3ExitInitializationMode" | optional |
| <code>vecubPreDoStep1</code> | At the beginning of "fmi3DoStep" | optional |
| <code>vecubPostDoStep2</code> | At the end of "fmi3DoStep" | optional |
| <code>vecubPreTask2</code> | Before calling a task | optional |
| <code>vecubPostTask2</code> | After calling a task | optional |
| <code>Terminate2</code> | During "fmi3Terminate" | mandatory |

This `VecubCallbacks1` pointer keeps a list of function pointers which could be used in order for plugin to grab information from VECU-BUILDER.

Available functionalities (callbacks) are:

- logging
- get symbol info for global variables, tasks (address, type)
- get task object
- get symbol accessor (reading and writing global variables)

```

typedef void          (__cdecl* VecubCallbackLog2)      (vecubString);
typedef void*        (__cdecl* VecubCallbackGetSymbolInfo2) (vecubString); // returns SymbolInfo*
typedef int          (__cdecl* VecubCallbackGetTask2)   (vecubString, Task*&);
typedef PtrSymbolAccess& (__cdecl* VecubCallbackGetSymbolAccessor2) (vecubString);

using VecubCallbacks2 = struct VecubCallbacks2 {
    VecubCallbackLog2      log2;
    VecubCallbackGetSymbolInfo2  getSymbolInfo2;
    VecubCallbackGetTask2   getTask2;
    VecubCallbackGetSymbolAccessor2  getSymbolAccessor2;
};

```

Fig. 5-12: function pointers

Example for Usage of Callbacks

The following code visualizes how callbacks can be used for Plugin Version 2.

```

// ===== plugin vers. 02 =====
DllExport vecub2Status __cdecl vecubFmi3EnterInit2(const VecubCall-
backs2* vecubCallbacks)
{
    if (!vecubCallbacks)
        return vecub2Status::vecubError;

```

The code defines the function `vecubFmi3EnterInit2` that takes a pointer to the structure `VecubCallbacks2` as input. It checks if the passed pointer `vecubCallbacks` is valid. If not, it returns `vecub1Status::vecubError`.

```

    // logging
    if (vecubCallbacks->log2)
    {
        vecubCallbacks->log2("user message");
    }

```

It checks if the `log2` function is defined in `vecubCallbacks` structure. If yes, it calls the function with the argument `user message`.

```

    // get symbol info
    SymbolInfo* symbol = static_cast<SymbolInfo*>(vecubCallbacks-
>getSymbolInfo2("factor1"));
    std::cout << "The symbol has address: " << std::hex << symbol-
>addr << std::endl;

```

It calls `getSymbolInfo2` function to obtain information about a symbol named `factor1`. The returned symbol is stored in the variable `symbol`. The address of the symbol is printed using `std::cout`.

```

    // get symbol accessor
    PtrSymbolAccess& symbolAccessor = vecubCallbacks->getSym-
bolAccessor2("factor1");
    VarTypes val = (*symbolAccessor).get();
    val.f64 = 5;
    (*symbolAccessor).set(val);

```

It calls `getSymbolAccessor2` function to obtain an accessor for the `factor1` symbol. The accessor is stored in the reference variable `symbolAccessor`. It reads a value of type `IntFloat64` from the symbol and assigns the value 5 to it. Then, it writes the value back to the symbol.

```

// get task object
Task* task{nullptr};
vecubCallbacks->getTask2("task_10ms", task);
task->setTNextCall(123);
if (task) {
    std::cout << "FOUND! task: " << " ~ " << task->getName() <<
std::endl;
}

```

It calls `getTask2` function to obtain a task object with the name `task_0ms`. The task object is stored in the pointer variable `task`. `setTNextCall` function is called on the task object with the value `123` as the argument. If the task object is valid, a message is printed with the task's name.

```

return vecub2Status::vecubOK;
}

```

It returns `vecub1Status::vecubOK`.

5.5.4 What a Plugin Can Do With Tasks V2

A plugin can access a task defined in VECU-BUILDER and can change different properties of a task at run-time.

As described in [Plugin Functions](#), you can use several functions. One of these functions is used to get a task. The used function pointer is `getTask2`, which returns a task type. For more details, see [Fig. 5-12](#)

Once the plugin access a task, it can manage through the task interface and the behavior of a task during run-time.

The interface of the task is available in Plugin template project through the definition of `task` class in `include/vecubTypes.h` file.

In this class, you can see specific functions of task class which manage i.e the task name, task period, or task priority.

6 Controlling VECU-BUILDER

6.1 Manual Interaction

You can operate VECU-BUILDER via the provided command and batch/shell scripts. For some user inputs, such as selecting a workspace directory, the tool may display dialogs.

6.2 Command-Line Interface

Besides the manual Interaction method, you can also operate VECU-BUILDER via a command-line interface (CLI). VECU-BUILDER is a CLI native application, and the command and batch/shell scripts allow manual interaction. To get more information about the Installation using CLI, see [Silent Installation of VECU-BUILDER](#).

The following arguments exist:

- new-project-path: Path where the workspace is to be created.
- no-dialogs: Suppress all dialogs and always select the default option.
- stop-on-success: Prevent automatic forwarding to the next stage (create workspace, import, build).
- version: Print the version information.
- h: Print list of all optional arguments.

To see all CLI optional arguments and their description

1. Open a command prompt on **Windows**

or

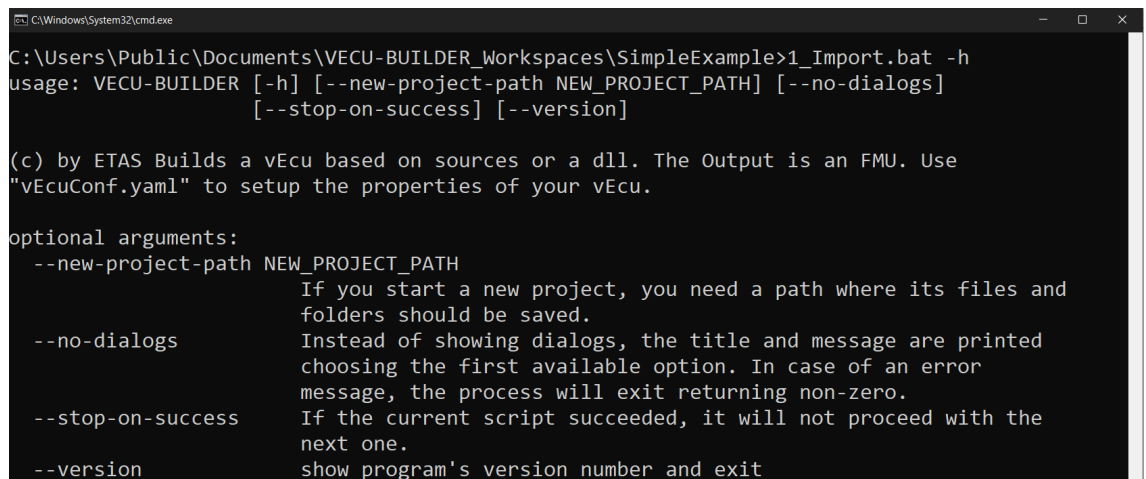
a terminal on **Ubuntu 20.04 LTS**.

2. Execute the following command:

`1_Import.bat -h` for **Windows**

or

`1_Import.sh -h` for **Ubuntu 20.04 LTS**.



```
C:\Windows\System32\cmd.exe
C:\Users\Public\Documents\VECU-BUILDER_Workspaces\SimpleExample>1_Import.bat -h
usage: VECU-BUILDER [-h] [--new-project-path NEW_PROJECT_PATH] [--no-dialogs]
                  [--stop-on-success] [--version]

(c) by ETAS Builds a vEcu based on sources or a dll. The Output is an FMU. Use
"vEcuConf.yaml" to setup the properties of your vEcu.

optional arguments:
  --new-project-path NEW_PROJECT_PATH
                        If you start a new project, you need a path where its files and
                        folders should be saved.
  --no-dialogs
                        Instead of showing dialogs, the title and message are printed
                        choosing the first available option. In case of an error
                        message, the process will exit returning non-zero.
  --stop-on-success
                        If the current script succeeded, it will not proceed with the
                        next one.
  --version
                        show program's version number and exit
```

Fig. 6-1: CLI optional arguments (Windows)

The CLI control method is ideal for integrating VECU-BUILDER into an automation pipeline. The CLI behaviour is the same as running the scripts manually: each script would call the next script to proceed through the stages of create a workspace, import, build. To change this behaviour, use `--stop-on-success`.

The following table gives an overview of which batch file uses which arguments:

| argument | CreateWorkspace | 1_Import | 2_Build |
|---------------------------------|-----------------|-----------------|-----------------|
| <code>--new-project-path</code> | Used (required) | Ignored | Ignored |
| <code>--no-dialogs</code> | Used (optional) | Used (optional) | Used (optional) |
| <code>--stop-on-success</code> | Used (optional) | Used (optional) | Ignored |
| <code>--version</code> | Used (optional) | Used (optional) | Used (optional) |
| <code>-h</code> | Used (optional) | Used (optional) | Used (optional) |

Tab. 6-1: Mapping of CLI arguments to scripts

To build the SimpleExample via two command lines

After creating the workspace, stop the process so that you can copy a specific YAML file into your workspace. Then trigger the import without `stop-on-success` and let it finish the build automatically.

1. Open a command prompt on **Windows**

or

a terminal on **Ubuntu 20.04 LTS**.

2. Navigate to the directory where the installer is located executing the following command:

```
cd %VECUBUILDER_HOME%
```

3. Execute the following command:

`CreateWorkspace.bat` on **Windows**

or

`CreateWorkspace.sh` on **Ubuntu 20.04 LTS**.

with the arguments

```
--new-project-path <destination>
```

```
--no-dialogs
```

```
--stop-on-success
```

where `<destination>` points to your workspace folder.

```

C:\Windows\System32\cmd.exe
C:\Program Files\ETAS\VECU-BUILDER\

```

Fig. 6-2: Workspace creation via CLI (Windows)

Note

A default YAML file is used in all newly created workspaces.
Project specific YAML file can be either prepared manually or in the previous step of your automation pipeline.

To use your project specific YAML file in this newly created workspace:

1. Execute the following command:

```
copy /y <source> <destination>.
```

```

C:\Windows\System32\cmd.exe
C:\Program Files\ETAS\VECU-BUILDER\

```

Fig. 6-3: Copy your project specific YAML file (Windows)

Note

The argument /y suppresses the prompt and thus overwrites the destination file.

To continue building your workspace:

1. Navigate to this new workspace by executing the following command:

```
cd <destination>.
```

2. Run the command:

```
1_Import.bat --no-dialogs for Windows
```

or

```
1_Import.sh --no-dialogs for Ubuntu 20.04 LTS.
```


6.3 Ubuntu 20.04 on WSL Command Line Interface

It is possible to create a new workspace on WSL Ubuntu 20.04 using the Ubuntu 20.04. Command Line Interface.



Note

Downloading dependencies or installing VECU-BUILDER only runs in WSL1, using VECU-BUILDER only runs in WSL2. Make sure that the WSL version matches the respective action. If necessary, you need to change the version.

For WSL1: `wsl --set-version Ubuntu-20.04 1`

For WSL2: `wsl --set-version Ubuntu-20.04 2`

1. Open PowerShell and set WSL Ubuntu 20.04 version to 2 using the following command:
2. Open Ubuntu 20.04 and make sure that the environment variables are set correctly using the following command:

```
wsl --set-version Ubuntu-20.04 2
```

```
env | grep -i vecu
```

```
~$ env | grep -i vecu
VECUBUILDER_EXAMPLES=/opt/ETAS/VECU-BUILDER/Examples_<version>/
VECUBUILDER_HOME=/opt/ETAS/VECU-BUILDER/<version>/
```

3. Change directory using the following command:
4. Create a new workspace using the following command:

```
cd $VECUBUILDER_HOME
```

```
./CreateWorkspace.sh --no-dialogs --new-project-path /opt/etas/VECU-BUILDER_Workspaces/SimpleExample/
```

Or

With installed Gnome Terminal workspace creation also works in Dialog Mode using the following command:

```
./CreateWorkspace.sh --new-project-path /opt/etas/VECU-BUILDER_Workspaces/SimpleExample/
```

- ⇒ A new workspace was created. For more details about workspace content, see [Workspace Content](#).

7 Debugging vECU

VECU-BUILDER provides useful functionalities to debug your vECU. It is possible to debug the vECU by using an Integrated Development Environment (IDE), such as Visual Studio Code or Visual Studio.

As the folder `<workspace>/vECU` is a CMake project, any IDE that can import CMake projects can be used for debugging.

During the Build stage, the debugging environment and batch/shell script files are created enabling you to enter a debugging session in just a few clicks.

You can use the `debug_hook` attribute, which can be enabled in the YAML file. vECUs built with this attribute enabled enter their instantiation and wait for a debugger to be attached by the user before continuing.



Note

The VECU-BUILDER debugging functionality is intended to be used for debugging of a single vECU within its workspace. If your vECU is integrated into a simulation, the `debug_hook` might be the best option for debugging.

The below table summarizes the possible combinations of build tool and debugger:

| | | Debugger | | | |
|------------|---------|--------------------|-------------|--------------|--------------------|
| | | VS Code | VS 2017 | VS 2019 | VS 2022 |
| Build tool | MinGW | recommended | unavailable | experimental | recommended |
| | VS 2017 | experimental | recommended | possible | possible |
| | VS 2019 | experimental | unavailable | recommended | possible |
| | VS 2022 | experimental | unavailable | unavailable | recommended |

Tab. 7-1: Debugging possibilities

Combinations marked as experimental, are neither tested nor supported and their use is solely your responsibility.

Among the recommended combinations, two are particularly recommended for use and are described in detail in the following chapters.

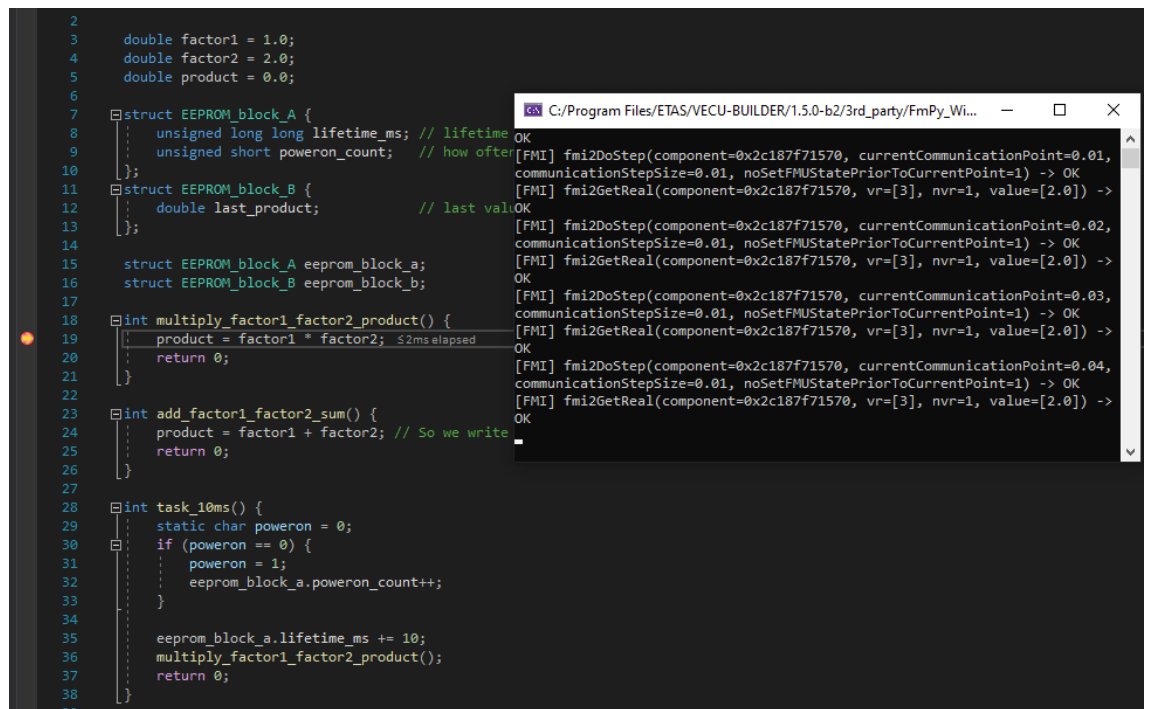
7.1 Debugging with Visual Studio 2019

This chapter describes how to debug vECU built with Visual Studio 2019 using Visual Studio 2019 as the debugger.

More information about Visual Studio 2019 can be found [here](#).

To debug with Visual Studio 2019

1. Navigate to your workspace.
2. Execute the `3b_StartDebugger.bat` file on **Windows** or `3b_StartDebugger.sh` on **Ubuntu 20.04 LTS**.
 - ⇒ The VS2019 debugger is invoked and loads the CMake project.
3. Navigate to where you want to start debugging and place a breakpoint there.
4. In the “Menu” tab click **Debug > Start Debugging (F5)**.
 - ⇒ FMPy is invoked and the debugger is attached.



```

2
3 double factor1 = 1.0;
4 double factor2 = 2.0;
5 double product = 0.0;
6
7 struct EEPROM_block_A {
8     unsigned long long lifetime_ms; // lifetime
9     unsigned short poweron_count; // how often
10 };
11 struct EEPROM_block_B {
12     double last_product; // last value
13 };
14
15 struct EEPROM_block_A eeprom_block_a;
16 struct EEPROM_block_B eeprom_block_b;
17
18 int multiply_factor1_factor2_product() {
19     product = factor1 * factor2; // 2ms elapsed
20     return 0;
21 }
22
23 int add_factor1_factor2_sum() {
24     product = factor1 + factor2; // So we write
25     return 0;
26 }
27
28 int task_10ms() {
29     static char poweron = 0;
30     if (poweron == 0) {
31         poweron = 1;
32         eeprom_block_a.poweron_count++;
33     }
34
35     eeprom_block_a.lifetime_ms += 10;
36     multiply_factor1_factor2_product();
37     return 0;
38 }

```

```

C:/Program Files/ETAS/VECU-BUILDER/1.5.0-b2/3rd_party/FmPy_Wi...
[OK]
[FMI] fmi2DoStep(component=0x2c187f71570, currentCommunicationPoint=0.01,
communicationStepSize=0.01, noSetFMUStatePriorToCurrentPoint=1) -> OK
[FMI] fmi2GetReal(component=0x2c187f71570, vr=[3], nvr=1, value=[2.0]) ->
OK
[FMI] fmi2DoStep(component=0x2c187f71570, currentCommunicationPoint=0.02,
communicationStepSize=0.01, noSetFMUStatePriorToCurrentPoint=1) -> OK
[FMI] fmi2GetReal(component=0x2c187f71570, vr=[3], nvr=1, value=[2.0]) ->
OK
[FMI] fmi2DoStep(component=0x2c187f71570, currentCommunicationPoint=0.03,
communicationStepSize=0.01, noSetFMUStatePriorToCurrentPoint=1) -> OK
[FMI] fmi2GetReal(component=0x2c187f71570, vr=[3], nvr=1, value=[2.0]) ->
OK
[FMI] fmi2DoStep(component=0x2c187f71570, currentCommunicationPoint=0.04,
communicationStepSize=0.01, noSetFMUStatePriorToCurrentPoint=1) -> OK
[FMI] fmi2GetReal(component=0x2c187f71570, vr=[3], nvr=1, value=[2.0]) ->
OK

```

Fig. 7-1: VS 2019 Debugger attached

7.2 Debugging with Visual Studio Code

This chapter describes how to debug vECU built with MinGW using Visual Studio Code as the debugger.

Prerequisites For Debugging with Visual Studio Code

It is obligatory to install the following packages in Visual Studio Code:

- Microsoft C/C++ Extension Pack

For Debugging in WSL Ubuntu with Visual Studio Code additionally install the following packages:

- C/C++ extensions for Visual Studio Code and WSL Ubuntu in Windows PC Host
- CMake extensions for Visual Studio Code and WSL Ubuntu in Windows PC Host
- CMAKE Tools extensions for Visual Studio Code and WSL Ubuntu in Windows PC Host
- gdb in Ubuntu WSL (see [Installing Dependent Software Packages.](#))

Without the installation of this package debugging is not possible.

Visual Studio Code requires some further extensions and will prompt you to install them by default. If you accept Visual Studio Code's suggestions, you should be good to go.

More information about Visual Studio Code can be found [here](#).

To debug with Visual Studio Code in Windows

1. Navigate to your workspace.
2. Right-click in your workspace and select **Open with Code**.
⇒ Visual Studio Code opens.
3. Navigate to where you want to start the debugging and place a breakpoint there.
4. Click **Start Debugging (F5)**.
5. In the menu panel on the left click **Run and Debug**.
⇒ FMPy is invoked and the debugger is attached.

To debug with Visual Studio Code in Ubuntu 20.04

1. Navigate to your workspace.
2. Start debugging using the following command:


```
$ ./3b_StartDebugger.sh
```
- ⇒ Visual Studio Code opens.
3. Navigate to where you want to start the debugging and place a breakpoint there.
4. In the menu panel on the left click **Run and Debug**.
5. Click **Start Debugging (F5)**.
- ⇒ FMPy is invoked and the debugger is attached.

To debug with Visual Studio Code in WSL

1. Check if gnome-terminal and gdb are installed. If not installed, see [Installing VECU-BUILDER on Ubuntu 20.04 for WSL](#).
2. Navigate to your workspace.
3. Start debugging using the following command:


```
./3b_StartDebugger.sh
```
- ⇒ Visual Studio Code opens.
4. Navigate to where you want to start the debugging and place a breakpoint there.
5. Click **Start Debugging (F5)**.
6. In the menu panel on the left click **Run and Debug**.
- ⇒ FMPy is invoked and the debugger is attached.

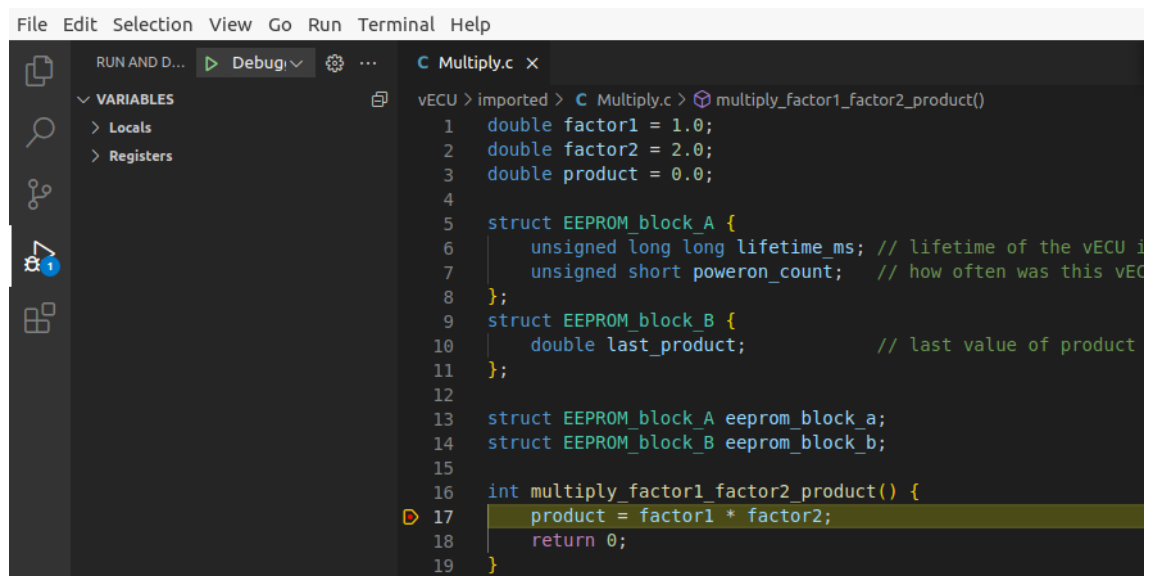


Fig. 7-2: VS Code Debugger attached

8 Troubleshooting

This chapter lists possible warning or error messages, their possible reasons and a possible solution to fix the issue.

8.1 CMake not found

```
(C) 2020-2023 ETAS GmbH. All rights reserved.
VECU-BUILDER <version>
#####
### Building sources of vECU                                     ###
#####
[09:15:24] 1 of 4: Reading config: vEcuConf.yaml
[09:15:24] 2 of 4: Creating Visual Studio Code debug configuration
[09:15:24] 3 of 4: Running scripts triggered through "before_build_sources"
                  - No script defined in the vEcuConf.yaml file
[09:15:24] 4 of 4: Compiling and linking
                  - SimpleExample.dll (Windows 64bit)

CMake not found.
For more details, please refer to the User Guide.
*** FAILURE ***
```

Fig. 8-1: CMake not found error

Possible Reason

A CMake installation is required and must be registered properly. ([Software Requirements for Windows 10](#)). This registry entry is used to locate the CMake installation, if it does not exist, the build fails.

It appears as if CMake was not installed or is not properly registered on your computer.

Possible Solution

Ensure the following:

- CMake is installed (version 3.15 or higher).
- Kitware and CMake keys exist in the Windows Registry.
- The CMake registry key `Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Kitware\CMake` contains the string value `InstallDir` pointing to the CMake installation path:

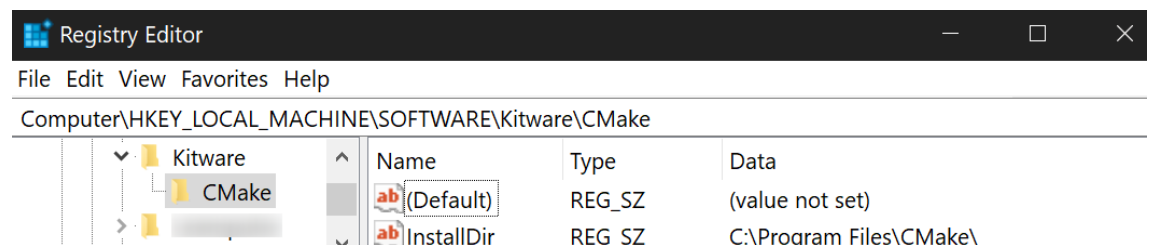


Fig. 8-2: Windows Registry with Kitware\CMake registry key

8.2 Notepad++ Does Not Open During Workspace Creation

Notepad++ is the recommended text editor to be used along with VECU-BUILDER. For it to work as intended, it must be installed and registered properly.

If Notepad++ does not open during the Workspace Creation stage, but Windows Notepad opens instead, it is either not installed at all or is not properly registered on your computer.

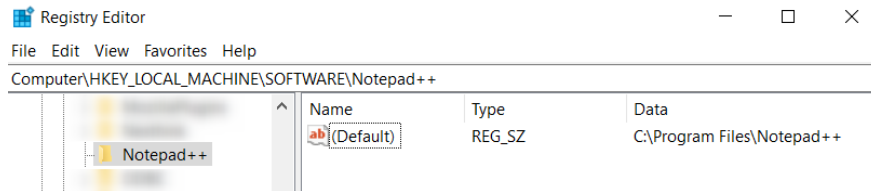
Possible Solution

Ensure the following:

- Notepad++ is installed.
- Notepad++ key exists in the Windows Registry.

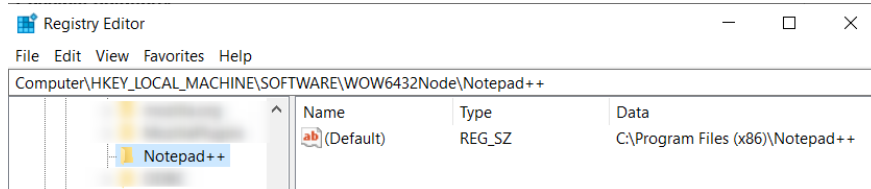
A. For 64-bit version:

- The Notepad++ registry key `Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Notepad++` contains the string value (Default) pointing to the Notepad++ installation path:



B. For 32-bit version:

- The Notepad++ registry key `Computer\HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Notepad++` contains the string value (Default) pointing to the Notepad++ installation path:



8.3 Some Breakpoints Not Being Hit

Possible Reason

Depending on your compiler configurations, the resulting vECU may be built so that some debugging information is not available. This may result in the debugger not being able to hit some breakpoints.

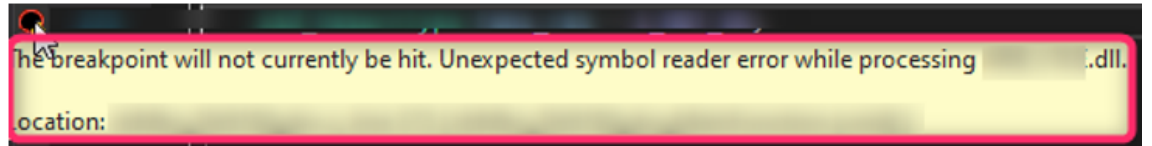


Fig. 8-3: Breakpoint not being hit

Possible Solution

In order to prevent such compiler optimization, include the following pragma statements:

- For MSVC compiler: `#pragma optimize("", off)`
- For MinGW compiler: `#pragma GCC optimize ("O0")`

8.4 (SymbolInfo.dll) The *.die File Is Too Large to Load

Possible Reason

The operating system does not provide sufficient amount of memory required to load the *.die file.

Possible Solution

Use a computer with sufficient amount of memory.

8.5 Windows Cannot Access Localhost While Using Sync Attribute in EEPROM

Possible Reason

EEPROM simulation feature requires entering the value of sync sub-attribute as UNC path.

If the defined location (e.g., C:/drive of your localhost) cannot be accessed during the vECU execution, the data defined by the sync sub-attribute cannot be used.

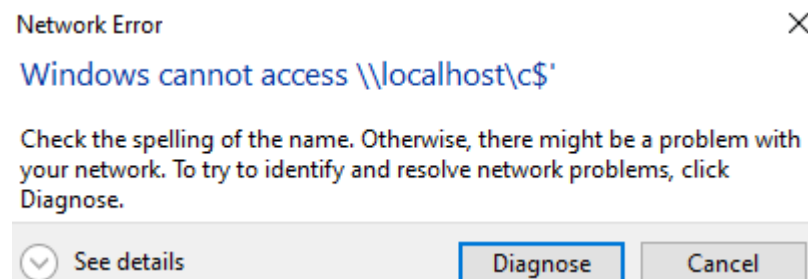


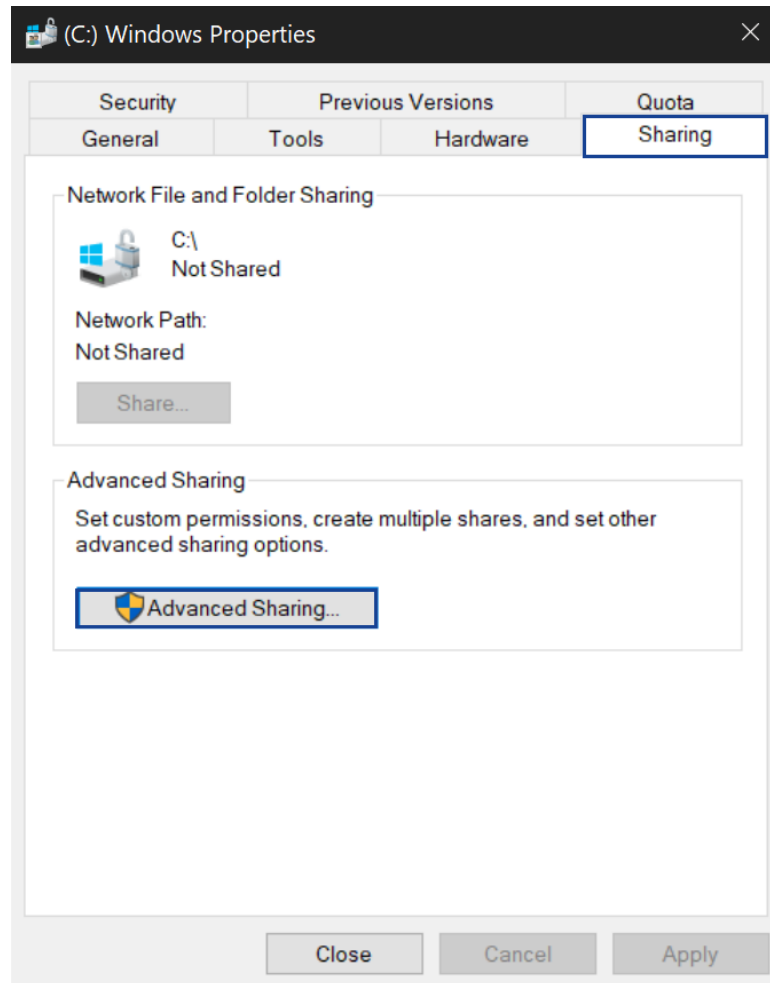
Fig. 8-4: Network Error - Localhost cannot be accessed

Possible Solution

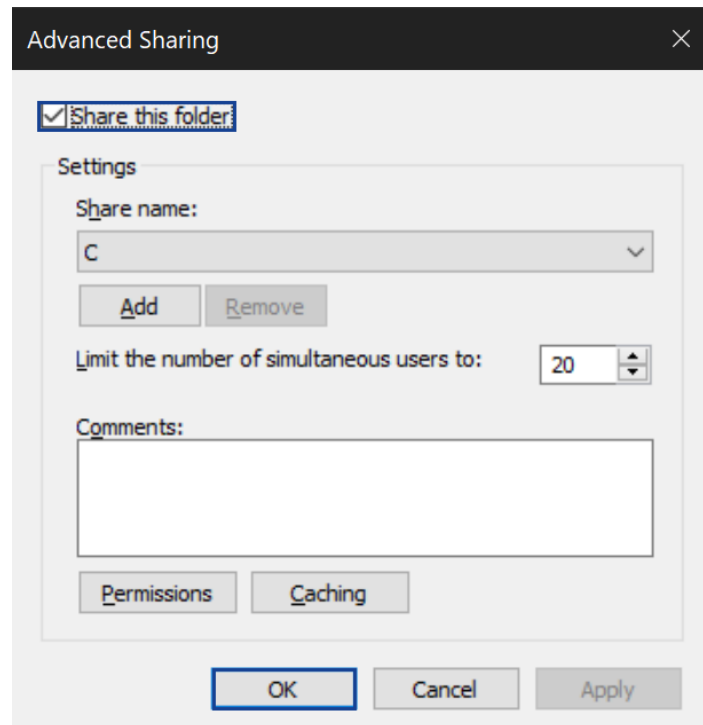
Setup the local share and obtain the UNC pathname.

To setup a local share:

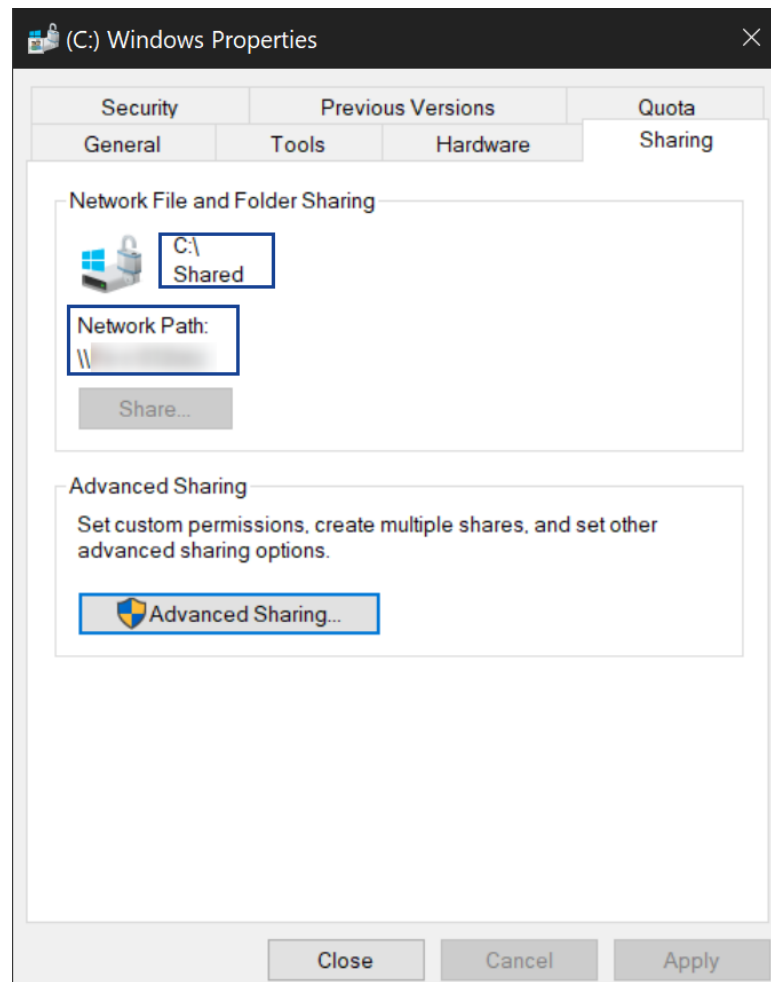
1. Navigate to the drive, you want to share. (e.g., (C:/drive))
2. Right-click in the drive and click **Properties**.
3. Click the **Sharing** tab.
4. Click **Advanced Sharing**. You will need Admin Rights to proceed.



5. Activate **Share this folder**.
6. Click **OK**.



⇒ The drive is now shared and the Network Path is displayed.

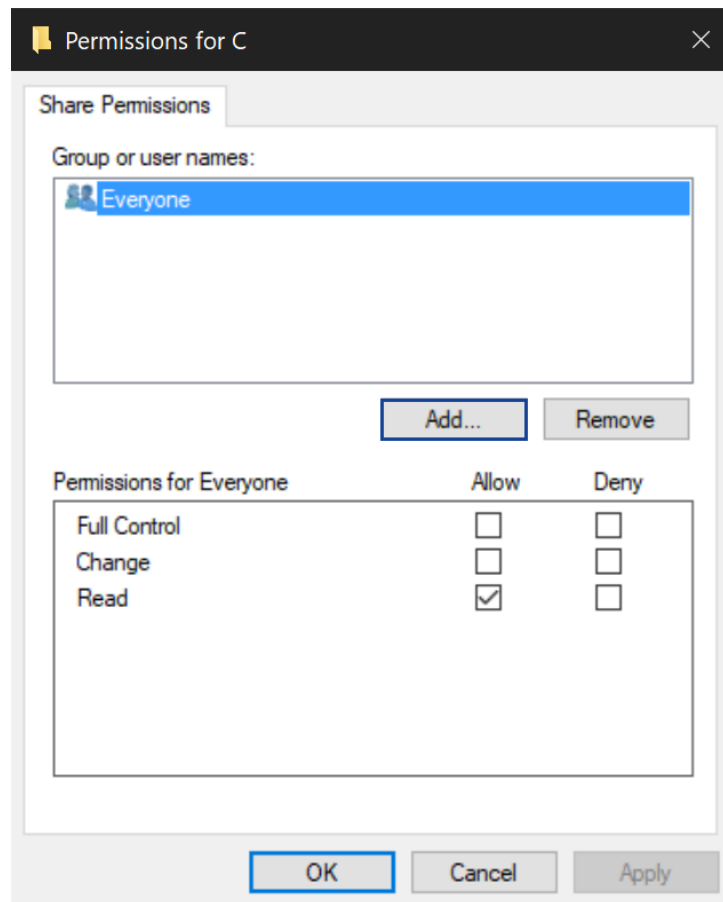


The user that will be logged in during the execution of the vECU, needs to be given full control permissions to the shared location.

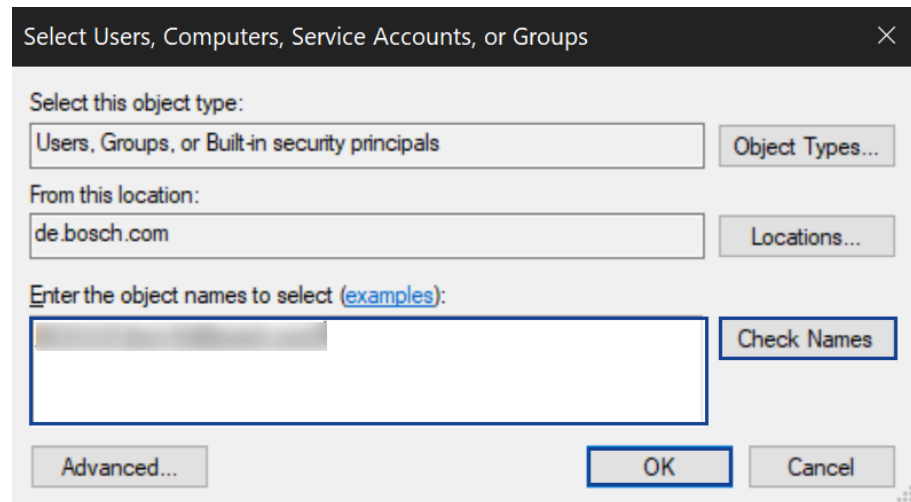
Per default, Windows will provide permissions to 'everyone'. The permissions should only be provided to the user, that will be logged in during the execution of the vECU. Therefore, the permissions shall to be changed for security reasons.

To change the permissions

1. Click **Advanced Sharing**. You might need Admin Rights to proceed.
2. Click **Permissions**.
3. Click **Add**.

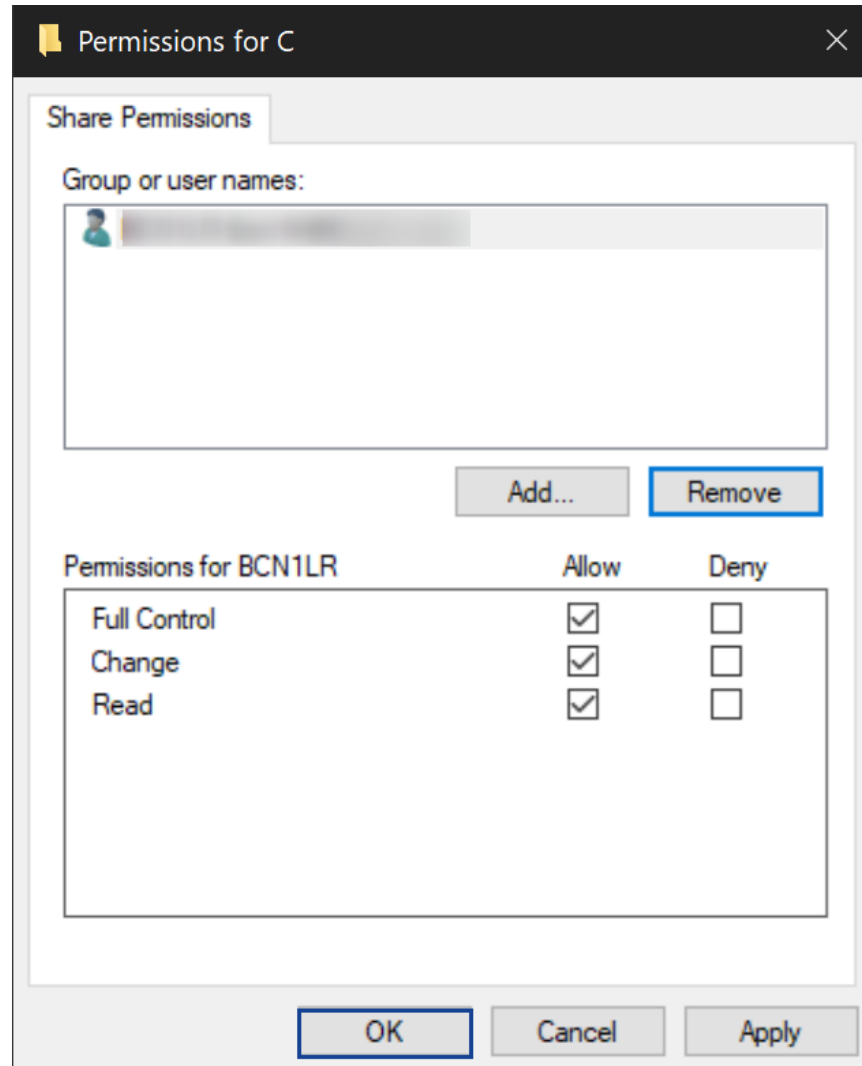


4. Enter the object name (username) to be selected.
5. Click **Check Names**.
6. Chose the displayed name.
7. Click **OK**.



8. Click the username to mark the entry.
9. Activate the permissions **Full Control** and **Change**.
10. To mark the entry, click **Everyone**.
11. To remove the permission for 'everyone', click **Remove**.

- ⇒ The group Everyone is removed and the selected user has now full permissions.



12. To confirm the **User Selection**, click **OK**.
13. To confirm the updated **Advanced Sharing properties**, click **OK**.
14. Close the **Properties** window.

8.6 Redirecting Function Calls Did Not Work as Expected

Possible Reason

The GNU compiler optimization level 2 (**-O2**) includes `inline-small-functions` which is incompatible with `redirect_function_calls`.

By default VECU-BUILDER uses the compile settings `RelWithDebInf`, which includes some optimizations. For gcc this would use the setting **-O2**, which includes `inline-small-functions`.

Possible Solution

Change the settings in `additional_compile_flags` to enable `redirect_function_calls`.

There are 3 ways to deactivate the optimization:

- A. **-O0**: Completely deactivates optimization. This has the advantage that the compiler time of user workspace decreases.
- B. **-O1**: Reduces the level of optimization from default 2 to 1.
- C. **-O2 -f-no-inline-small-functions**: Keeps optimization to level 2 but only disables the special optimization with `-f-no-inline-small-functions`.

For more details, see [Options That Control Optimization](#).

8.7 License Check Failed

```

C:\WINDOWS\system32\cmd.exe

(C) 2020-2023 ETAS GmbH. All rights reserved.
#####
### Building FMU                                     ###
#####
[16:58:38] 1 of 6: Reading config: vEcuConf.yaml
[16:58:39] 2 of 6: Running scripts triggered through "before_build_fmus"
                - No script defined in the vEcuConf.yaml file
[16:58:39] 3 of 6: Building inputs, outputs, parameters, tasks
License check failed! For more details, please refer to the User Guide.

```

Possible Reason

- The LiMa installation is corrupt.
- LiMa might not reach the license server.

Possible Solution

- Reinstall VECU-BUILDER described in [Installation on Windows 10](#) and [Installation on Ubuntu 20.04 LTS](#) or contact [Technical Support](#).
- Check network settings to get a connection to the license server.

8.8 Building Sources Failed

Possible Reason

In some cases, building sources fails with various error messages. To save time, CMake uses caches, e.g. a link to the build-tool is stored.

Possible Solution

To fix a broken CMake cache, delete the cache and rebuild the sources.

1. Navigate to the `vECU` folder.
2. Delete everything except the `imported` folder.
3. Rebuild the sources using `2_Build.bat` on **Windows** or `2_Build.sh` on **Ubuntu 20.04 LTS**.

If the build fails due to CMake reason you can find more details in `build/-log/build_cmake.log` file.

8.9 Indentation Errors in YAML File

It is possible that indentation errors may occur which are difficult to detect.

Possible Solution

To check for indentation errors in yaml file, use a YAML Checker. There are online tools available. You can search for 'yaml checker' in any search engine. To search, you also can use this [link](#). Follow the instructions given by the selected YAML Checker.

8.10 Failed to Parse Symbols

It can happen that symbol/debug information is missing in the binary. The error message below treats the missing debug information during build process when `build_mode` is `import_compiled`.

```
(C) 2020-2024 ETAS GmbH. All rights reserved.
VECU-BUILDER
#####
### Building FMU                                     ###
#####
[11:23:02] 1 of 6: Reading config: vEcuConf.yaml
[11:23:02] 2 of 6: Running scripts triggered through "before_build_fmus"
                  - No script defined in the vEcuConf.yaml file
[11:23:02] 3 of 6: Building inputs, outputs, parameters, tasks
Failed to parse symbols.
For more details, please refer to the User Guide.
```

Possible Reason

The error occurs due to some mishandling of DLL/SO when `build_mode` in yaml file is set to `import_compiled`.

Possible Solution

- If the used `build_tool` is one of the Visual Studio compile versions (like 16 2016 or 17 2022), make sure that besides the DLL loaded, there shall also exist a mandatory PDB file. The source location for DLL & PDB is given by yaml settings `import_external_compiled_vecu` and `get_updates_from`.
- If the `build_tool` used is **MinGW Makefiles for Windows** or **Unix Makefiles for Linux**, only the DLL/SO is required. Make sure that the DLL/SO mandatorily contains debug information. A DIE file is created locally when building FMUs runs. Afterwards also `SymbolDetails.txt` is created.
- Make sure the `build_tool` yaml settings match the compiler used to build the DLL:
 - If the imported DLL was built with the GNU compiler from MinGW, make sure that `build_tool` is **MinGW Makefiles** (`build_tool: MinGW Makefiles`).
 - If the imported DLL was built with a Visual Studio compiler, make sure that `build_tool` is **Visual Studio xx xxxx** (`build_tool: Visual Studio xx xxxx`).

For more information see [import_external_compiled_vecu](#) in Configuration chapter.

8.11 Could not Load the vECU Binary

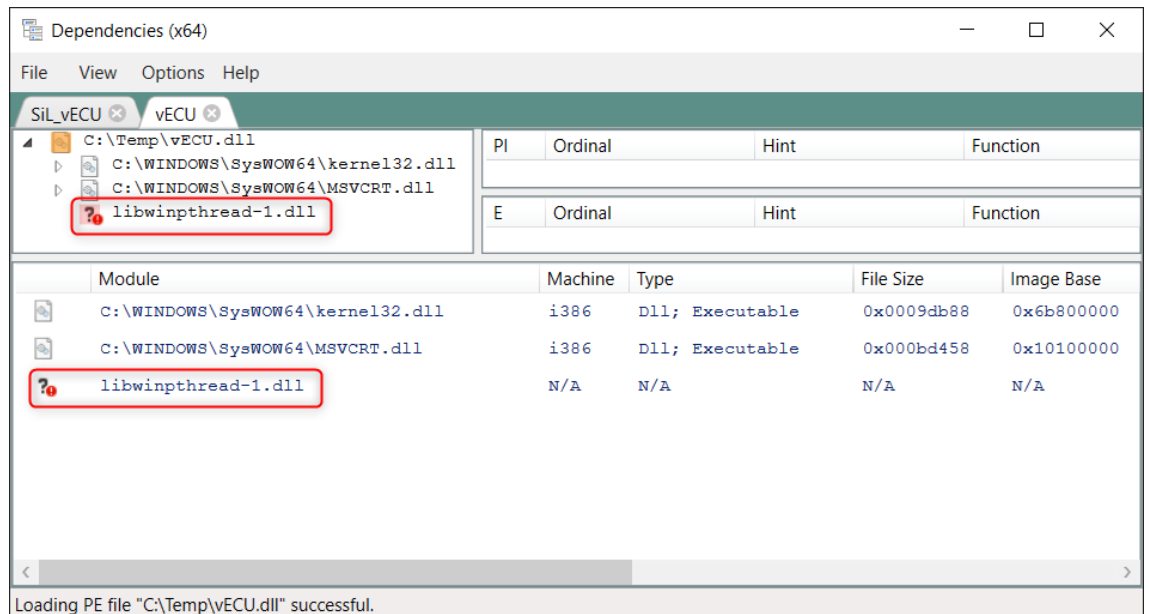
During the vECU FMU build the `SymbolDetails.txt` variables are updated with the initial values by loading of the `vECU.dll` binary and reading the values from RAM. In cases where the `vECU.dll` cannot be loaded, the `SymbolDetails.txt` variables are given the default value 0 and the user is informed with an appropriate message in the `build_fm.log` file. In this case, the FMU build is possible, but the generated FMU execution will not work due to the same `vECU.dll` loading problem as during the build.

Possible Reason

One or more dynamic libraries, required for the `vECU.dll` execution are missing / could not be found by the OS.

Possible Solution for Windows:

Check by loading the `vECU.dll`, which dynamic libraries are missing using the open source SW "[Dependencies](#)".



Provide the missing info about the dynamic library to the system using the `vECUConf.yaml` options:

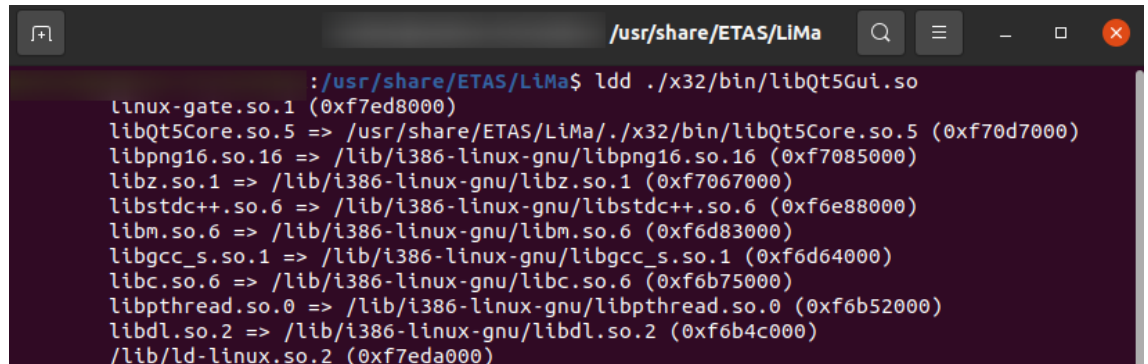
Option 1: (most recommended): Add the library to the resources folder of the `vECU.fmu` using "`additional_resources`".

Option 2: Provide the corresponding path to the missing library using "`environment_variables`".

Possible Solution for Ubuntu 20.04:

To check the dependencies in Linux run `ldd` command on a `*.so` file.

Example: `ldd libQt5Gui.so`



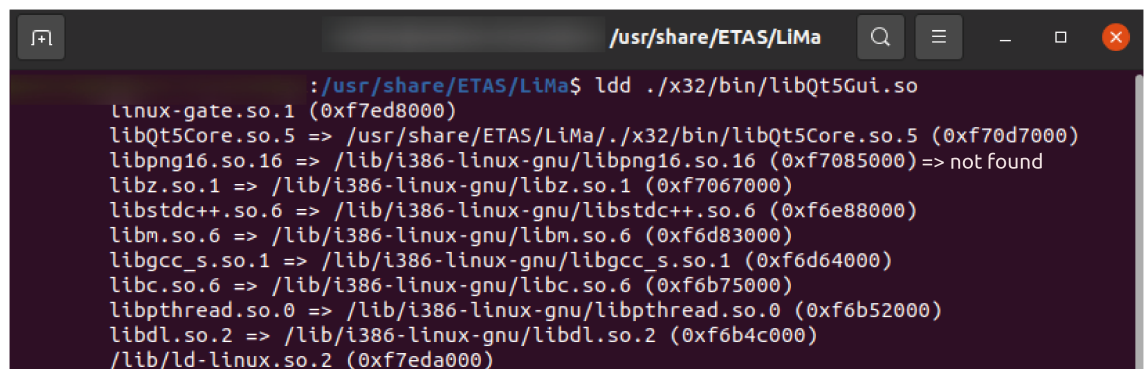
```

/usr/share/ETAS/LiMa$ ldd ./x32/bin/libQt5Gui.so
linux-gate.so.1 (0xf7ed8000)
libQt5Core.so.5 => /usr/share/ETAS/LiMa/./x32/bin/libQt5Core.so.5 (0xf70d7000)
libpng16.so.16 => /lib/i386-linux-gnu/libpng16.so.16 (0xf7085000)
libz.so.1 => /lib/i386-linux-gnu/libz.so.1 (0xf7067000)
libstdc++.so.6 => /lib/i386-linux-gnu/libstdc++.so.6 (0xf6e88000)
libm.so.6 => /lib/i386-linux-gnu/libm.so.6 (0xf6d83000)
libgcc_s.so.1 => /lib/i386-linux-gnu/libgcc_s.so.1 (0xf6d64000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf6b75000)
libpthread.so.0 => /lib/i386-linux-gnu/libpthread.so.0 (0xf6b52000)
libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xf6b4c000)
/lib/ld-linux.so.2 (0xf7eda000)

```

Fig. 8-5: Check dependencies in Linux

This command will give you the list of the dependencies. If one file is missing this command will give a "not found" message.



```

/usr/share/ETAS/LiMa$ ldd ./x32/bin/libQt5Gui.so
linux-gate.so.1 (0xf7ed8000)
libQt5Core.so.5 => /usr/share/ETAS/LiMa/./x32/bin/libQt5Core.so.5 (0xf70d7000)
libpng16.so.16 => /lib/i386-linux-gnu/libpng16.so.16 (0xf7085000) => not found
libz.so.1 => /lib/i386-linux-gnu/libz.so.1 (0xf7067000)
libstdc++.so.6 => /lib/i386-linux-gnu/libstdc++.so.6 (0xf6e88000)
libm.so.6 => /lib/i386-linux-gnu/libm.so.6 (0xf6d83000)
libgcc_s.so.1 => /lib/i386-linux-gnu/libgcc_s.so.1 (0xf6d64000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf6b75000)
libpthread.so.0 => /lib/i386-linux-gnu/libpthread.so.0 (0xf6b52000)
libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xf6b4c000)
/lib/ld-linux.so.2 (0xf7eda000)

```

Fig. 8-6: "Not found" message

One way to tell to an application where to search for its SO dependencies is to set `LD_LIBRARY_PATH` to the location where these SO resides.

8.12 Skipping Plugin

```
[WARNING] Skipping plugin plugin_template_vers.01.dll  
[OK] Loading plugin plugin_template_vers.02.dll  
[FMI] fmi3InstantiateCoSimulation(instanceName="Fmu30", instantiation)
```

Possible Reason

The FMI version set in yaml file configuration does not fit to the plugin version.

Possible Solution

Check the selected FMI version in the yaml file and make sure that it corresponds to the correct and required plugin version.

- If FMI version is set to 2 in yaml file, then the plugin version shall be 1.
- If FMI version is set to 3 in yaml file, then the plugin version shall be 2.

If there is no correspondence between FMI version and plugin version, change it accordingly.

9 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 offers trainings for its products:

www.etas.com/academy



ETAS Headquarters

ETAS GmbH

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