
	<p>Professorship Computer Engineering Automotive Software Engineering Prof. Dr. Dr. h. c. Wolfram Hardt Dr. René Bergelt</p>	 TECHNISCHE UNIVERSITÄT CHEMNITZ
	CE Virtual ECU - Manual	11-2024

Contents

1. Introduction	1
2. Prerequisites	1
2.1. Setting up on Windows	2
2.1.1. Installing .NET 8.0	2
2.1.2. Installing C compiler	2
2.1.3. Installing Visual Studio Code	4
2.2. Setting up on Linux	4
2.2.1. Installing .NET 8.0	4
2.2.2. Installing C compiler	4
2.2.3. Installing Visual Studio Code	5
2.3. Setting up on MacOS	5
2.3.1. Installing .NET 8.0	5
2.3.2. Installing C compiler	5
2.3.3. Installing Visual Studio Code	6
3. Developing for the CE Virtual ECU	7
3.1. Setting up the project template	7
3.1.1. Additional steps on Windows	9
3.1.2. Additional steps for Linux	9
3.1.3. Additional steps for MacOS	9
3.2. Compiling and Running your ECU program	10
4. Simulating CAN Bus Communication	12
4.1. Windows: NtCan	12
4.2. Linux: SocketCAN	13
4.3. MacOS	15

1. Introduction

This manual will guide you in setting up the CE Virtual ECU, which is a software-based simulator of the ECU platform (based on the STMicroelectronics SPC560P microcontroller) used in Unit 1 & 2 in the Automotive Software Engineering practical. By providing you with this simulator, the Professorship Computer Engineering allows you to practice embedded C programming with regards to the specific ECU used in the practical as well as to prepare for the respective practical sessions at home.

2. Prerequisites

Before you can program for the Virtual ECU you need to setup a corresponding development environment on your computer. In general, the following things need to be installed:

- .NET 8.0 Runtime (needed to run the emulator itself)

- C compiler and debugger
- C IDE (we will use Visual Studio Code)
- CAN bus driver (only needed for CAN simulation), see Section 4

Based on the target computer system (i.e. Windows, Linux or MacOS) different steps for installing the prerequisites might be required. Please refer to the section which describes the procedure for the operating system you have.



Currently, the Virtual ECU is only supported on **64-bit systems**. So make sure that your computer has a 64-bit CPU and is running a 64-bit operating system. Usually, this should almost always be the case for most modern systems.

2.1. Setting up on Windows

The Virtual ECU can be run on Windows 10 and later. Previous versions of Windows might work but are neither officially supported nor tested.

2.1.1. Installing .NET 8.0

Make sure that the .NET 8.0 Runtime or SDK for **x64** is installed on your PC. The installer can be downloaded from <https://dotnet.microsoft.com/en-us/download/dotnet/8.0>. At the time of writing, the latest version is [8.0.10](#).

Alternatively, if you are using a package manager, usually there should be a package for .NET 8.0 you can install by invoking the corresponding command as follows:

winget:

```
winget install dotnet-runtime-8
```

Chocolatey¹:

```
choco install dotnet-8.0-runtime
```

2.1.2. Installing C compiler

We will use GCC² as C compiler to compile the code for the Virtual ECU. As GCC is not available by default on Windows we need to install a corresponding development environment which contains it. While there are other options, for this guide we will use MSYS2 as it is very easy to setup and to be kept up-to-date.



If you already have a recent version of GCC & GDB installed (maybe from other sources), you can skip this step. However, make sure to update the path in the project template's settings.json accordingly.

Basically, it is sufficient to follow the basic setup guide available on the homepage of MSYS2³ as this includes the installation of GCC. However, one additional package for GDB needs to be installed as well. For clarity's sake, the relevant steps from the website are replicated here. If you need more details, please refer to MSYS2's complete official documentation.

¹<https://chocolatey.org>

²<https://gcc.gnu.org>

³<https://www.msys2.org>

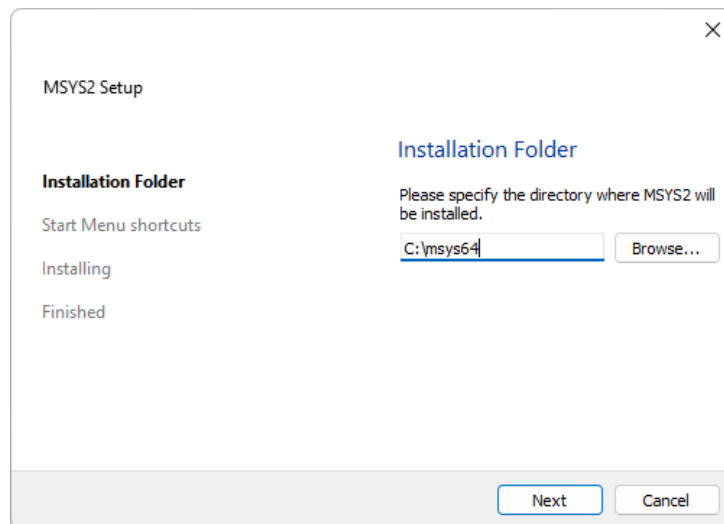


Figure 1: Specifying the installation path for MSYS2⁴

1. Download the latest installer from [msys2.org](https://www.msys2.org)
At the time of writing this is [msys2-x86_64-20240727.exe](#)
2. Run the installer. Installing MSYS2 requires 64 bit Windows 10 or newer.
3. Leave the installation folder as is; see Figure 1 for reference
4. When the installation is done, click Finish.
5. Now MSYS2 is ready for you and a terminal for the UCRT64 environment will launch as shown in Figure 2.
6. In order to install GCC & GDB please run the following command in this terminal and confirm the download and installation:

```
pacman -S mingw-w64-ucrt-x86_64-gcc mingw-w64-ucrt-x86_64-gdb
```

7. Once the installation has succeeded you can close the terminal

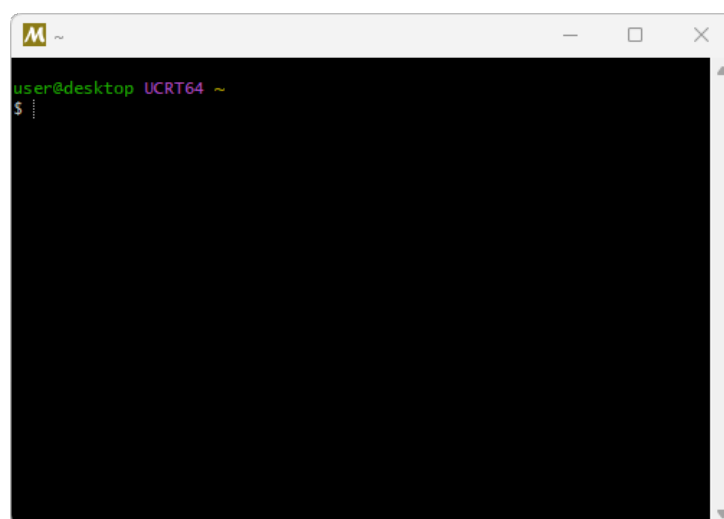


Figure 2: The MSYS2 terminal

⁴<https://www.msys2.org>

2.1.3. Installing Visual Studio Code

As the provided template project for the Virtual ECU is a Visual Studio Code workspace (and since you will use Visual Studio Code in later units of the practical as well), we will now set it up to build programs for the Virtual ECU. Obviously, if you already have an installation of VS Code you can skip this step, just make sure that you are using one of the latest versions.

For this, download the latest installer from <https://code.visualstudio.com> by clicking on “Download for Windows” and execute it.

Alternatively, if you have a package manager available, run the corresponding command to install VS Code. For example:

winget:

```
winget install -e --id Microsoft.VisualStudioCode
```

Chocolatey:

```
choco install vscode
```

Congratulations, you have successfully set up the development environment for the Virtual ECU and you can continue with Section 3 to see how you can implement programs for the Virtual ECU for preparing for the ASE practical units 1 and 2.

2.2. Setting up on Linux

The CE Virtual ECU should work on most linux distributions based on Debian 9+ (Stretch and later), Ubuntu 16.04+ as well as Fedora 30+.



Based on the actual Linux distribution you use, the actual steps or commands to execute might differ.

2.2.1. Installing .NET 8.0

Browse to <https://dotnet.microsoft.com/en-us/download/dotnet/8.0> and follow the setup instructions for your type of linux distribution. Only the “.NET Runtime” is required to run the Virtual ECU. For many distributions .NET can be installed over the official package manager. For instance, if your distribution is using **apt** you can run the following to set everything up:

```
apt install dotnet-runtime-8.0
```

2.2.2. Installing C compiler

Usually, most Linux distributions already come with gcc and gdb. You can check this by running:

```
gcc --version
```

If you get a similar output as follows, you are already finished with this step:

```
gcc (Ubuntu 13.2.0-23ubuntu4) 13.2.0
Copyright (C) 2023 Free Software Foundation, Inc.
```

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

If the command is not recognized, please refer to the documentation of your distribution to install the gcc compiler toolchain as well as gdb.

2.2.3. Installing Visual Studio Code

Please follow the official guide on installing Visual Studio Code on Linux: <https://code.visualstudio.com/docs/setup/linux>.



While you can technically use any IDE or editor to program the Virtual ECU, we currently only provide a project template for Visual Studio Code since it is available cross-platform. If you want to use your own toolchain, please refer to the tasks.json and launch.json files in the Visual Studio Code project template to extract the corresponding compiler and gdb commands (see Section 3)

2.3. Setting up on MacOS

While MacOS is technically supported, we cannot offer any extended support or guidance due to lack of Apple hardware for testing. Due to this, the Virtual ECU has not been tested on the ARM-based products of Apple (Apple M1 etc.). As an alternative, there is always the possibility to use a virtualized Linux system on your Mac in order to execute the Virtual ECU, e.g. by using VirtualBox⁵. You can then follow the steps in Section 2.2 to set up the virtualized Linux accordingly.

If you follow this guide to setup the Virtual ECU on your Mac device, any feedback from you would be much appreciated.

2.3.1. Installing .NET 8.0

Browse to <https://dotnet.microsoft.com/en-us/download/dotnet/8.0> and follow the setup instructions for MacOS. Only the ".NET Runtime" is required to run the Virtual ECU. Make sure to download the correct package based on your system architecture (i.e. either Arm64 if you have an Apple Silicon based computer or x64 if it is based on Intel CPUs).

2.3.2. Installing C compiler

While gcc & gdb can generally be installed on OSX, we will setup the build environment using the clang compiler which comes preinstalled. To check if it is set up correctly, you should run the following in a terminal window:

```
clang --version
```

If you get a similar output as follows, you are already finished with this step:

```
Apple clang version 13.1.6 (clang-1316.0.21.2.5)
Target: x86_64-apple-darwin-21.1.0
```

⁵<https://www.virtualbox.org>

```
Thread model: posix  
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

If the command is not recognized, please refer to Apple's documentation on how to setup clang.

2.3.3. Installing Visual Studio Code

Please follow the official guide on installing Visual Studio Code on MacOS: <https://code.visualstudio.com/docs/setup/mac> .

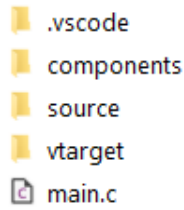


Figure 3: Folder structure of the template project

3. Developing for the CE Virtual ECU

3.1. Setting up the project template

Once you have installed all pre-requisites you are ready to use Visual Studio Code to implement and build programs for the CE Virtual ECU. For this, please download the zipped template project which is provided to you where you downloaded this manual. Once extracted, you should have the same folder structure and files as shown in Figure 3. This is basically the same project structure of the SPC5 Studio IDE used in the practical with the actual hardware.

After launching Visual Studio Code, click on File >> Open Folder... and navigate to the extracted folder of the template project and confirm. When the workspace has been opened, you should see the main window similar to what is shown in Figure 4.

On the left side, the “Explorer” window shows you the structure of the project as shown in Figure 5a. If it has not been opened automatically, click on the “main.c” file in the project explorer which contains the entry point of the ECU’s program. If you are asked to install the “C/C++ Extension Pack” when opening the “main.c” file as shown in Figure 5b, please do so. Alternatively, make sure that the “C/C++ Extension Pack” is installed and up-to-date through VS Code’s extension manager (see Figure 5c).

To finalize the project setup, please make sure to follow the instructions of the following subsections based on your platform:

Windows: see Section 3.1.1, Linux: see Section 3.1.2, MacOS: see Section 3.1.3.

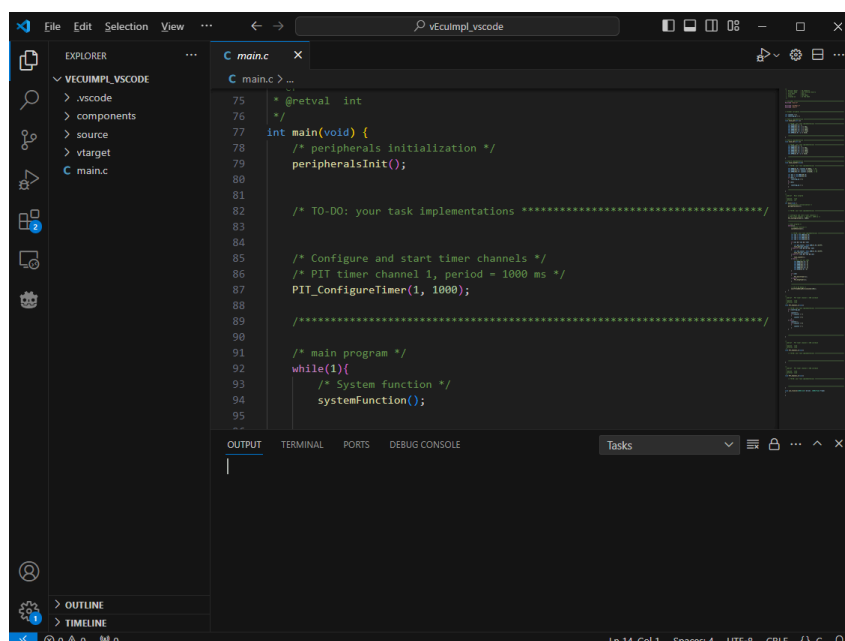
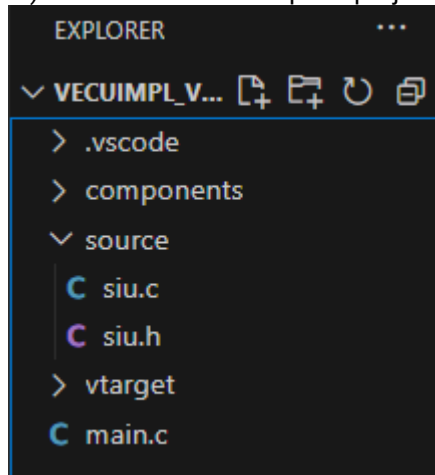
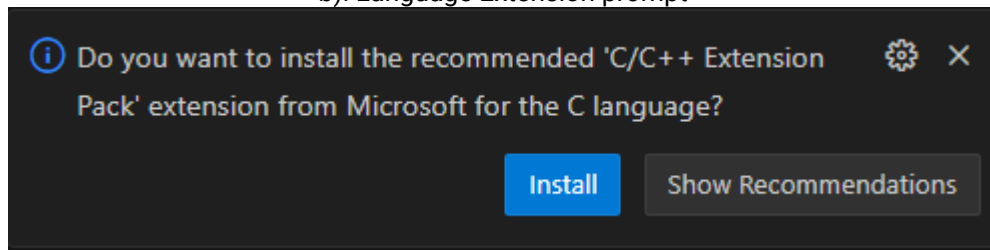


Figure 4: Main window of the Visual Studio Code IDE

a): Structure of the template project



b): Language Extension prompt



c): C/C++ Extension Pack

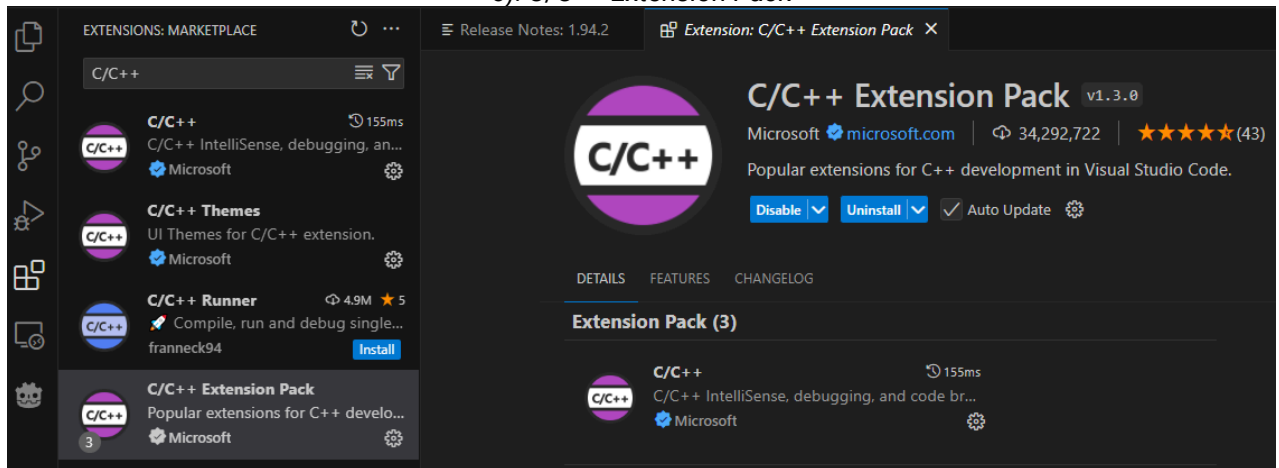


Figure 5: Views of Visual Studio Code

3.1.1. Additional steps on Windows

If you changed the default installation path of MSYS2 or want to use a different source for GCC and GDB, please set the correct path in the file `.vscode/settings.json` in the variable `build_tools_directory`.

Congratulations, the project is now correctly set-up for compiling and running the code on the CE Virtual ECU which is provided as part of the template project.

3.1.2. Additional steps for Linux

Please set the correct path in the file `.vscode/settings.json` in the variable `build_tools_directory` which points to the location of compiler and debugger. **Usually**, you can just set it to an empty value (i.e. `""`) since the compiler tools are globally accessible:

```
{
  "vecu":
  {
    "build_tools_directory": ""
  }
}
```

In addition, in order to be able to run the Virtual ECU interface for debugging you need to mark it as executable. To do this, run the following command with the appropriate permissions in the project's root folder:

```
chmod +x vtarget/linux/vECU.UI
```

Congratulations, the project is now correctly set-up for compiling and running the code on the CE Virtual ECU which is provided as part of the template project.

3.1.3. Additional steps for MacOS

Please set the correct path in the file `.vscode/settings.json` in the variable `build_tools_directory` which points to the location of compiler and debugger. **Usually**, you can just set it to an empty value (i.e. `""`) since the compiler tools are globally accessible:

```
{
  "vecu":
  {
    "build_tools_directory": ""
  }
}
```

The MacOS version of the Virtual ECU emulator is not part of the default project template. Consequently, you need to download the correct version based on your platform (ARM64 or x64) from the place where the Virtual ECU was provided to you (e.g. Opal). The emulator is provided as a zip file which contains both the appropriate vtarget folder and a customized Visual Studio Code configuration. Basically, you just need to extract the zip package in the root folder of the template project overwriting any existing files.

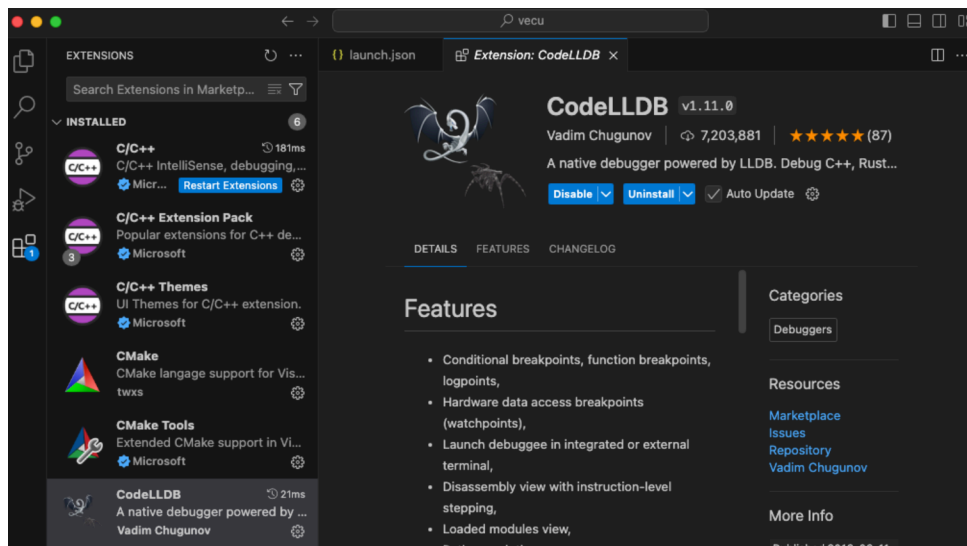


Figure 6: The CodeLLDB extension needed on MacOS

The .vscode folder in the zip archive is a hidden directory in MacOS. Make sure to nonetheless copy its contents. If you did it correctly, the “Run and Debug” task of the project should be prefixed with “[MacOS]” afterwards, i.e. “[MacOS] Launch with Virtual ECU” (Cf. Figure 8.)

In addition, in order to be able to run the Virtual ECU interface for debugging you need to mark it as executable. To do this, run the following command with the appropriate permissions in the project’s root folder:

```
chmod +x vtarget/osx/vECU.UI
```

Now, we need to install an extension to be able to debug the program using LLDB. For this switch to Visual Studio Code’s Extensions tab and search for the extension “CodeLLDB” and install it.

Finally, we need to relax the security policies on Visual Studio Code for it to run code which is not signed by a registered Apple Developer since neither your ECU program nor the Virtual ECU come with such a certificate. Do this by navigating to MacOS’s System Preferences > Security & Privacy and adding Visual Studio Code to the list of excluded apps as shown in Figure 7.

Congratulations, the project is now correctly set-up for compiling and running the code on the CE Virtual ECU which is provided as part of the template project.

3.2. Compiling and Running your ECU program

In order to **build and run** the program, either press “F5” or switch to the “Run and Debug View” (see Figure 8) and click on the green triangle next to “Launch with Virtual ECU”. If this is successful your program will be automatically *flashed* to the CE Virtual ECU which automatically opens in a new window. Please notice that it resembles the look of the front panel of the hardware ECU which is used in the ASE practical, see Figure 10 for reference. As with the real ECU you need to turn on the Virtual ECU by switching the power switch at the bottom to on (i.e. from 0 to —). By doing so after building the template project, the *Onboard LED* in the top right corner should start blinking to indicate that the code is indeed running.

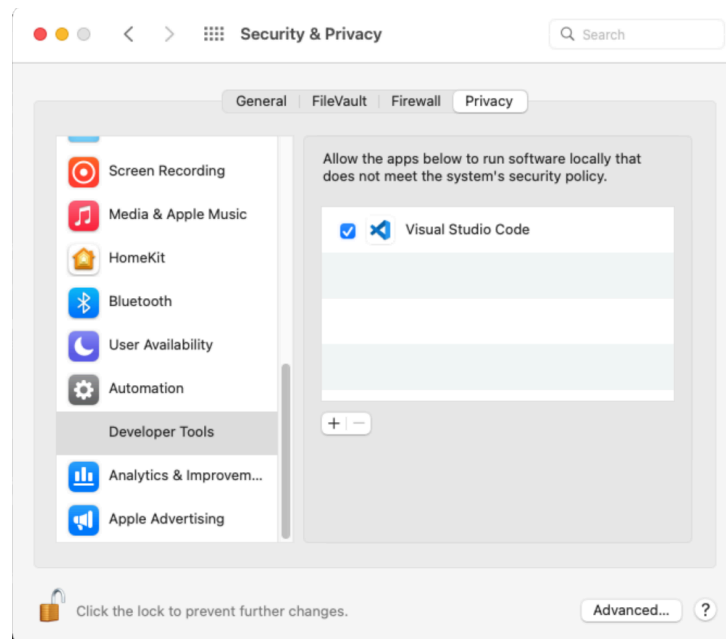


Figure 7: Allow Visual Studio Code to run unsigned code

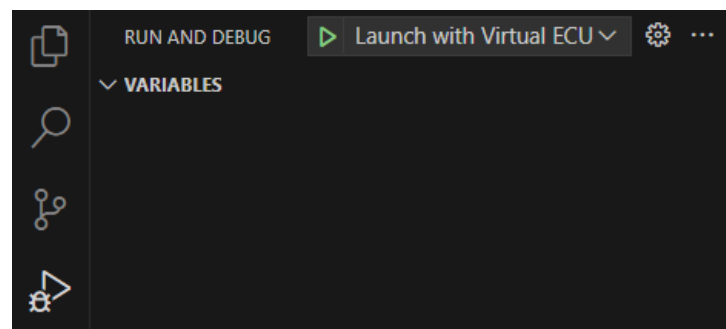


Figure 8: VS Code's Run and Debug View

If there are any errors in your code they will be shown in the problems view similar to Figure 9. In addition, the errors will be highlighted in the code itself. Fix the shown errors before trying to rebuild the program.

Close the Virtual ECU, by either clicking on the window's X button or by turning the Power Switch back to off.

You can now start to implement programs which access the buttons, LEDs and additional sensors. You can follow the normal task description of Unit 1 as the CE Virtual ECU is programmed in exactly the same way as the hardware ECU used in the practical (this also applies to the need

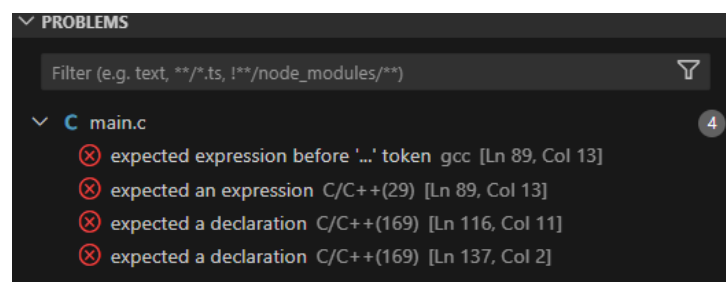
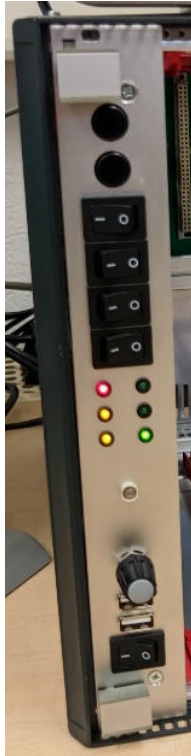


Figure 9: VS Code's Problems View

a): The Hardware ECU's front panel



b): Window of the Virtual ECU

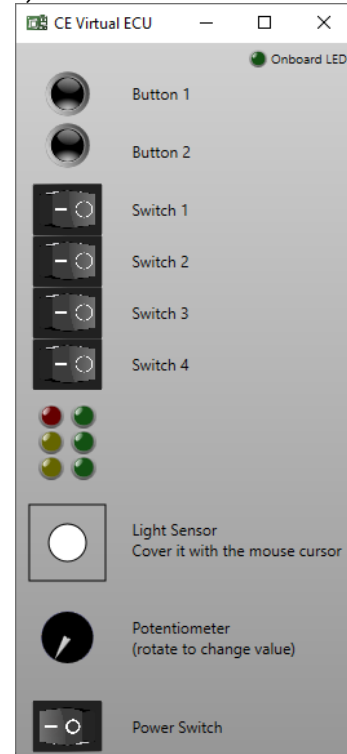


Figure 10: Comparison of actual hardware used in the practical and Virtual ECU

to correctly configure the pins as inputs or outputs). In general, you should only make changes in **main.c** and the files which are in the *source* directory (e.g. **siu.h**, **siu.c**, **can.h**, **can.c**).



If you prepared source code using the Virtual ECU **at home** you can bring these files (**main.c**, **siu.h**, **siu.c**, **can.h**, **can.c**) to the practical and copy the contents into the actual hardware ECU project of SPC 5 Studio.

4. Simulating CAN Bus Communication

In order to develop CAN communication without CAN hardware (e.g. to prepare Unit 2 of the practical) we will use a Virtual CAN driver which simulates a CAN network on your PC. The tools used for this differ by platform, so please follow the instructions in the section based on your operating system.

4.1. Windows: NtCan

We will use the Virtual CAN from esd electronics GmbH which can be downloaded and used for free. The Virtual CAN driver is part of esd's CAN SDK which you can download from <https://esd.eu/en/products/can-sdk>⁶. When installing the CAN SDK please make sure to tick both **.NET Class Library for managed code development** and **Virtual CAN FD device driver** as shown in .

The CAN-enabled Virtual ECU will connect to the ESD virtual CAN after start-up. Consequently, you can configure and use the ECUs CAN controller in your code as described in the task description of Unit 2.

⁶Direct download link: https://esd.eu/fileadmin/esd/software/apps/can/CAN_SDK.exe

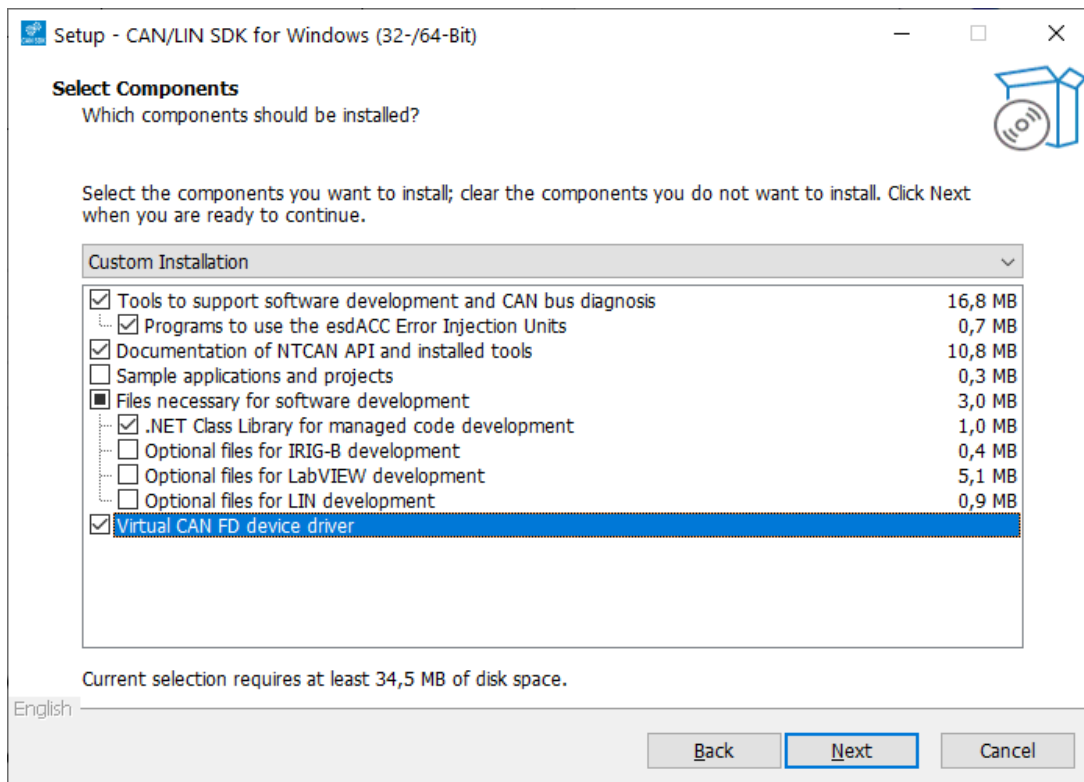


Figure 11: Components of the esd CAN SDK which need to be installed

You can visualize the CAN communication by using the CAN monitor tool from esd which is called "CANreal". It should have been installed by the SDK setup. You can start it through your start menu (see Figure 12). After starting CANreal the main window should be visible as shown in Figure 13. You can connect to the virtual CAN bus which is also used by CE Virtual ECU by selecting the **Net**: 42 - CAN_VIRTUALFD and using **BTR**:500.0 kbps as bitrate. After clicking "Start" the messages on the CAN bus will be shown in the window's list. In addition, you can send CAN messages (which can then be received by the Virtual ECU) by using the bottom controls. See the official documentation for an in-depth description of features: https://esd.eu/fileadmin/esd/docs/manuals/en/CANreal_Manual_en_33.pdf

4.2. Linux: SocketCAN

Most Linux distributions now come with CAN support built into the kernel in the form of SocketCAN⁷. Please make sure that you are using a Linux distribution with a CAN-enabled kernel ((at least 2.6.30).

We will test the CAN communication without actual CAN hardware. Consequently, we need to enable the "virtual can" kernel module of SocketCAN and at a vcan socket as follows (execute with appropriate permissions):



Figure 12: Windows 10 start menu entry for esd CANreal

⁷<https://github.com/linux-can/>

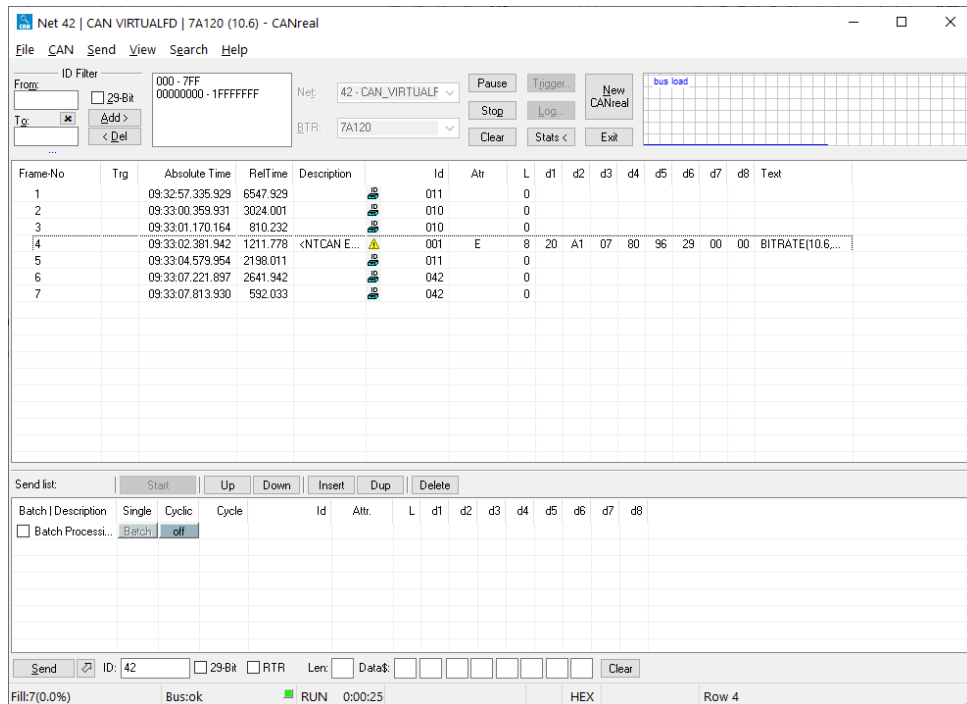


Figure 13: Main window of ESD CANreal, connected to the Virtual CAN Bus (Net ID: 42)

```
modprobe vcan
sudo ip link add name vcan0 type vcan
sudo ip link set up vcan0
```

In order to test sending and receiving CAN messages you can use the SocketCAN tools *cansend* and *candump*. These are part of the *can-utils* package which is available for most distributions. For example if your distribution is using apt, you can install it as follows:

```
apt install can-utils
```

Starting the CAN-enabled Linux version of the Virtual ECU automatically connects to the SocketCAN interface *vcan0*. Consequently, you can configure and use the ECUs CAN controller in your code as described in the task description of Unit 2. You can then test if the CAN code you wrote for the Virtual ECU is working correctly by using the *can-utils*.

Use *candump* to show all messages which are sent over the virtual CAN bus *vcan0* by executing the following command in a terminal:

```
candump vcan0
```

Use *cansend* to send messages to the virtual CAN bus *vcan0* as follows:

```
# send a standard frame message with ID 011 and no content
cansend vcan0 011
# send a standard frame message with ID 011 and content bytes 1 and 2
```

```
cansend vcan0 011#0102
# send an extended frame message with ID 011 and content bytes 1 and 2
cansend can0 00000001AC7#0102
```

4.3. MacOS

As of now, unfortunately there is no free virtual CAN driver available for Mac OS. We are working on a solution. For now, you can resort to using a virtualized Linux if you want to test the CAN communication.