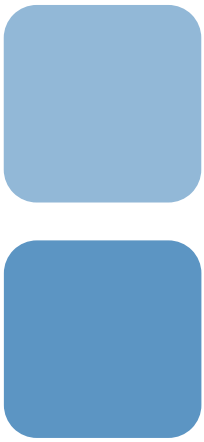
.NET on F&S Boards

Version 0.8
(2024-05-14)



Untere Waldplätze 23
D-70569 Stuttgart
Germany

Phone: +49(0)711-123722-0
Fax: +49(0)711-123722-99

About This Document

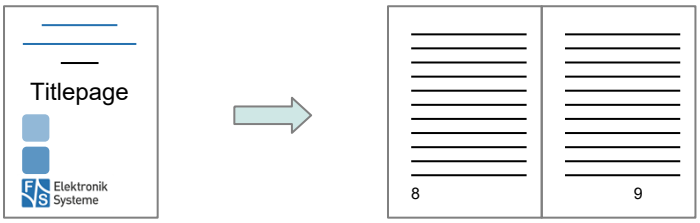
This document describes how to run .NET applications on F&S Boards.

Remark

The version number on the title page of this document is the version of the document. It is not related to the version number of any software release! The latest version of this document can always be found at http://www.fs-net.de.

How To Print This Document

This document is designed to be printed double-sided (front and back) on A4 paper. If you want to read it with a PDF reader program, you should use a two-page layout where the title page is an extra single page. The settings are correct if the page numbers are at the outside of the pages, even pages on the left and odd pages on the right side. If it is reversed, then the title page is handled wrongly and is part of the first double-page instead of a single page.



Typographical Conventions

We use different fonts and highlighting to emphasize the context of special terms:

`File names`

Menu entries

`Board input/output`

`Program code`

`PC input/output`

`Listings`

`Generic input/output`

`Variables`

History

Date	V	Platform	A,M,R	Chapter	Description	Au
2021-07-19	0.1	ALL	A	ALL	Initial version	PG
2021-07-23	0.2	ALL	M	ALL	Adapted formatting of document to F&S CI	HF
2021-07-26	0.3	ALL	A	6.3	Add VS2019 remote debugging chapter	PG
2021-07-27	0.4	ALL	M	ALL	Correct some typos, Remove false quotation marks in tasks example.	PG
2023-04-13	0.5	ALL	A,M	3,5,6	Added guide to create basic Hello World application, added command needed when installing debugger on board.	TG
2023-04-21	0.6	ALL	M	6.2	Fix pathes to match the FS-vscode-remote-debug-config Fix sshd_conf to sshd_config Add info about debugging not working on .NET7 arm	PG
2023-12-12	0.7	ALL	A,M	4.3,5.3,6.3.6.5	Add dotnet packages to Yocto Adapting code to run Avalonia on Linux and Windows Remote debugging in Visual Studio 2022	BS
2024-05-14	0.8	ALL	A,M	1,3,6.5.2,6.5.3,7,8	Demo App “FusDotnetDemo” Enabling RDP Copy files to board with PowerShell-Script	BS

V	Version
A,M,R	Added, Modified, Removed
Au	Author

Table of Contents

1 Introduction

This document describes how to run .NET application on F&S-Linux boards.

To access Linux hardware like I²C or SPI in .NET you need additional libraries which are released and maintained by Microsoft at

https://github.com/dotnet/iot

We released the demo application “FusDotnetDemo” on Github, an app to showcase some of the possibilities of .NET on Linux with F&S boards:

https://github.com/FSEmbedded/FusDotnetDemo

FusDotnetDemo is built using Avalonia UI and the main scope is to give some examples on how to access hardware interfaces in Linux with .NET by using the aforementioned libraries.

This document assumes basic knowledge of using Linux on F&S boards. For a detailed introduction please see the *Linux on F&S Boards.pdf* from the document section of your F&S board at

https://www.fs-net.de/

2 System Requirements

The .NET images need a lot of disk space, so make sure your flash memory is big enough:

Buildroot

Image type	Image size
Ubifs Image (Nand Flash)	210 MB
Ext4 Image (eMMC)	480 MB

Yocto

Image type	Image size
Ubifs Image (Nand Flash)	
Ext4 Image (eMMC)	

3 Tested Software Versions

The Software used in this document has been tested with the following versions.

Software	Version
Development Machine	F_S_Development_Machine-Fedora_30_V2.0 F_S_Development_Machine-Fedora_35_V1.0
Buildroot fsimx6 fsimx8	 fsimx6-B2021.10.1 fsimx8-B2021.06.1
Yocto	-
.NET	SDK 6.0.201 .NET Runtime 6.0.3 SDK 7.0.203 .NET Runtime 7.0.5
VSCode	1.65.2 1.77.1
VSDebugger	17.0.10413.12
Visual Studio 2019	16.10.4
VSRemoteDebugger (VS2019)	1.3
Visual Studio 2022	17.9.2

4 Compiling the .Net Images

F&S supports the build environments Buildroot and Yocto to build the system software.

This chapter describes how to build a root file system with preinstalled .NET binaries, using Buildroot or Yocto.

For a detailed description how to setup and use the build environments, please see the document *Linux on F&S Boards* chapter *Compiling the System Software*.

4.1 Prequisites

Note

For now, we will only describe how to modify an existing build to add .NET support. If there is enough interest in this matter, we will add recipes to build .NET images fully automatic.

You can download the .NET binaries from the official Microsoft website.

Make sure to download the right OS (Linux) and architecture (Arm32 for i.MX6 and Arm64 for i.MX8 based boards).

If you want to compile the code directly on the board, you will have to download the SDK. This however needs a lot of disc space, so make sure your board has enough flash memory available.

For most cases the .NET Runtime should be sufficient.

Copy the Binaries to your development machine.

4.2 Compiling the .NET images with Buildroot

1. Get the latest F&S-Buildroot release and execute the `setup-buildroot` script to install Buildroot to your development machine. Follow the instructions.
2. Build the respective defconfig of your machine. For example run:
   ```
   make fs<YOUR_MACHINE>_wayland_defconfig
   ```
   in your Buildroot main directory.
3. Open the configuration menu in your buildroot directory
   ```
   make menuconfig
   ```
4. Activate the ICU package at
   `Target packages -> Libraries -> Text and terminal handling -> icu`
5. Build buildroot
   ```
   make -j4
   ```
6. Create the directory
   `output/target/usr/share/dotnet-runtime/`
   and copy the previously downloaded .NET binaries to it (the complete content of the archive).
7. Create the file
   `output/target/etc/profile.d/dotnet.sh`
   and add the following content:

   ```
   #!/bin/sh
   export PATH=$PATH:/usr/share/dotnet-runtime/
   ```

```
export DOTNET_ROOT=/usr/share/dotnet-runtime/
```

This will export the path to the .NET installation each time you log in.

8. Build buildroot again:
   `make -j4`
9. The build output can be found at
   `output/images/`

4.3 Compiling the .NET images with Yocto

You have to add dotnet-core to your Yocto sources. You can find these on GitHub, follow this link for more information:

https://github.com/RDunkley/meta-dotnet-core

In your development machine, change directory to the sources for your build:

```
cd /path/to/your/release/build/yocto-fus/sources
```

Get dotnet from GitHub:

```
git clone https://github.com/RDunkley/meta-dotnet-core
```

Now you have to add the desired packages to your local.conf:

```
vi /path/to/your/release/build/yocto-fus/build-MACHINE-
DISTRO/conf/local.conf
```

Add following lines:

PREFERRED_VERSION_dotnet-core = "7.0.11"

CORE_IMAGE_EXTRA_INSTALL += "dotnet-core vsdbg "

You can change the dotnet-version to your preferred one.

In bblayers.conf you have to set the directory for your dotnet-recipes:

```
vi /path/to/your/release/build/yocto-fus/build-MACHINE-
DISTRO/conf/bblayers.conf
```

Add this line at the end of the file:

BBLAYERS += " ${BSPDIR}/sources/meta-dotnet-core "

Now, change directory to yocto-fus:

```
cd /path/to/your/release/build/yocto-fus/
```

Setup your build environment:

```
. setup-environment build-MACHINE-DISTRO
```

This command will create your image, it will only add the dotnet-core and vsdbg to your existing build:

```
bitbake fus-image-std
```

5 Executing .NET applications on F&S boards

This chapter describes how to execute .NET applications on F&S boards.

5.1 Running an Hello World Application

1. Install the .NET images to your board. You will need to install kernel, device tree and root filesystem. The different ways of how to install the images are described in the document *Linux on* F&S Boards chapters Image Download and Image Storage.

2. To Create a Basic 'Hello World' Application you can use this command, which will create an Application named MyApp based on .Net 7.

```
dotnet new console -o MyApp -f net7.0
```

3. Publish your application as linux-arm for i.MX6/7 boards and linux-arm64 for i.MX8 boards. Use the *–no-self-contained* flag to exclude the runtime binaries from your build

```
dotnet publish -r linux-arm -o bin\linux-arm\publish --no-self-contained
```

4. Boot Linux and transfer your .NET application files to the board. You can transfer them via network using the tftp command or use an USB stick. See *Linux on F&S Boards* chapter *Using the Standard System and Devices.*

5. Execute the .NET applications DLL using

```
dotnet /path/to/your/application.dll
```

5.2 Running an Avalonia Application

You can develop your application on your Windows Computer. To use Avalonia UI, you first have to install it to your development PC:

```
dotnet new install Avalonia.Templates
```

Create a new project with Avalonia UI:

```
dotnet new avalonia.app -o MyApp
```

If you use Visual Studio, you can skip these two steps and instead install Avalonia as a NuGet-package. Afterwards you can create a new project using Avalonia as template.

To run the new App on Linux, some adaptions have to be made to Program.cs :

```
using Avalonia.Media;
```

```
public static AppBuilder BuildAvaloniaApp()
…
.With(new FontManagerOptions() { DefaultFamilyName = "Liberation Sans" });
```

With Avalonia, it is possible to develop Apps that run on multiple platforms. If you want your software to run on Windows and Linux, you can automatically set the right font for your system:

```
public static AppBuilder BuildAvaloniaApp()
        {
            FontManagerOptions options = new();

            if (OperatingSystem.IsLinux())
            {
                options.DefaultFamilyName = "Liberation Sans";
            }
            // No need to set default for Windows
            return AppBuilder.Configure<App>()
                        .UsePlatformDetect()
                        .LogToTrace()
                        .With(options)
                        .UseReactiveUI();
        }
```

Choose a font that is available on Linux. If you need ReactiveUI for your program to work, add it to your AppBuilder.

To run your software on Linux, follow these steps:

1. Publish your application as linux-arm for i.MX6/7 boards and linux-arm64 for i.MX8 boards. Use the *–no-self-contained* flag to exclude the runtime binaries from your build

```
dotnet publish -r linux-arm -o bin\linux-arm\publish --no-
self-contained
```

2. Boot Linux and transfer your .NET application files to the board. You can transfer them via network using the tftp command or use an USB stick. See *Linux on F&S Boards* chapter *Using the Standard System and Devices.*

3. Execute the .NET applications DLL using

```
dotnet /path/to/your/application.dll
```

6 Remote debugging .NET apps on F&S Boards

You can use Visual Studio Code to program and compile your .NET applications as usual, but if you want to debug your application while running on the F&S board, some additional preparations are needed.

Note

As of this writing, remotely debugging .NET 7 apps in linux-arm environments is unreliable and may cause the process to exit prematurely. This issue is under investigation. .NET 6 apps that target linux-arm and .NET 7 apps that target linux-arm64 are unaffected.

6.1 Installing VSDebugger to the board

Download the VSDebugger

For i.MX6/7 from

https://vsdebugger.azureedge.net/vsdbg-17-0-10413-12/vsdbg-linux-arm.tar.gz

For i.MX8 from

https://vsdebugger.azureedge.net/vsdbg-17-0-10413-12/vsdbg-linux-arm64.tar.gz

Note

Remote Debugging was tested with version 17-0-10413-12. There might be a newer version available. You can test it by editing the download string.

The vsdebugger for arm needs about 104 MB of disk space.

If your board has enough flash memory you can install the VSDebugger like the dotnet Runtime binaries:

Buildroot

1. Create the directory `output/target/usr/share/dotnet-runtime/vsdbg-linux` and extract the downloaded files to it. (Make sure there is no additional sub directory)
2. Rebuild buildroot and copy the new rootfs to the board.´
3. Once the Debugger is installed on the Board you will need to give the 'vsdbg' file (by default located in /usr/share/dotne-runtimet/vsdbg-linux/) execute Permissions. This is done with the following command:

```
chmod +x vsdbg
```

You can also copy the files to an SD card or USB stick and mount it at the board. You will have to adapt some paths later on then.

6.2 Enabling root access via SSH

VSCode needs root access via SSH for remote debugging.

To allow root to login via SSH with no password set, some preparations are needed.

Please note that this should only be done for development purposes!

Buildroot

1. Mount the rootfile system read-writeable

```
mount -o remount,rw /
```

2. Edit the file /etc/ssh/sshd_conf using the vi editor

```
vi /etc/ssh/sshd_config
```

3. Edit the following lines (also remove the hashes):

```
(press 'i' to enter edit mode)
#PermitRootLogin prohibit-password -> PermitRootLogin yes
#PermitEmptyPasswords no           -> PermitEmptyPasswords yes
(press 'Esc' to exit edit mode)
(type ':wq' to save and quit)
```

4. Restart the ssh daemon

```
/etc/init.d/S50sshd restart
```

5. Set an IP address on the board. You can either use DHCP running the command

```
udhcpc
```

or set it per hand with the command

```
ifconfig eth0 up <YOUR.BOARD.IP.ADRESS>
```

You should now be able to log into root per SSH without entering a password.

6.3 Visual Studio Code

6.3.1 Configuring VSCode

In order to launch the VSDebugger on the board, you will have to create or edit the launch.json file. If it does not already exist you will be asked to create it when clicking on the *Run and Debug* tab at the side bar.

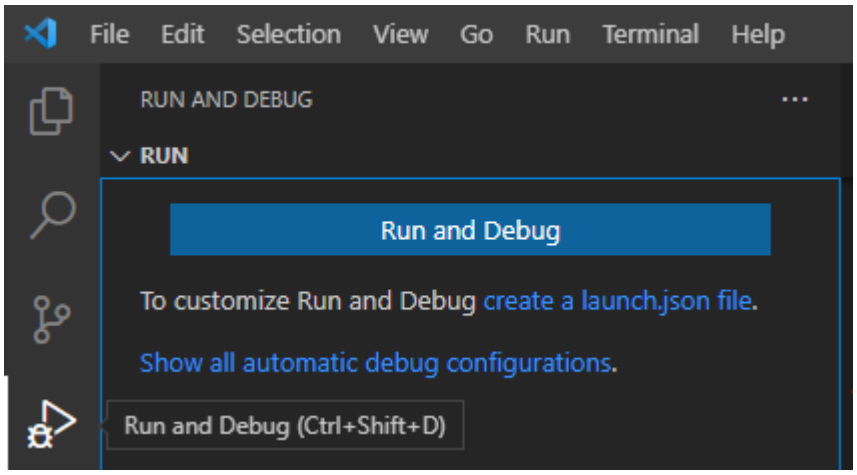


Figure 1 Creating a launch.json file

Download the FS-vscode-remote-debug-config repository from

https://github.com/FSEmbedded/FS-vscode-remote-debug-config

And copy/replace the `files launch.json, tasks.json` and `settings.jsons` to the `.vscode/` directory of you project.

Edit the following red marked lines in `settings.jsons` if necessary:

- Set your board IP address here. Use ifconfig to show your boards IP address.

```
"FuS.boardIp": "10.0.0.103",
```

- Change this path if you installed the VSDebugger at a different location

```
"FuS.debuggerPath": "/usr/share/dotnet-runtime/vsdbg-linux/vsdbg"
```

- Change this path if you want to place your application to the flash instead of ram only (e.g to /opt)

```
"FuS.TargetPath": "/tmp",
```

- Change this to arm64 for i.MX8 boards

```
"FuS.boardArch": "arm",
```

Add this to `launch.json`:

```
            "env": {
                "DISPLAY": ":0",
                "XDG_RUNTIME_DIR": "/run/user/0"
              },
```

6.3.2 Start debugging in VSCode

Start debugging by clicking the Run and Debug button at the Run and Debug tab.

Your .NET application should be built, transferred to the board. The application should be started and stop, if you have set a break point.

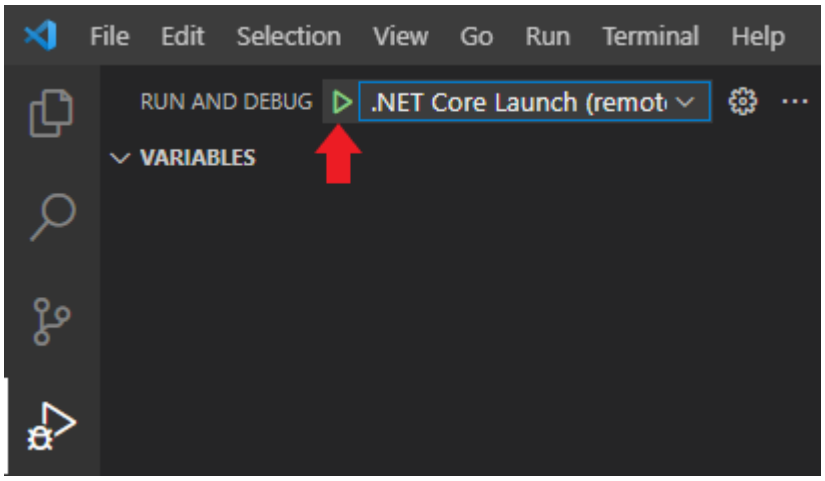


Figure 2 Start Debugging

6.4 Visual Studio 2019

There are no official solutions for remote debugging in Visual Studio 2019 yet.

You could either use Visual Studio Code to debug your application, or try community projects like

https://github.com/radutomy/VSRemoteDebugger

The VSRemoteDebugger needs some additional preparations to work with F&S boards.

6.4.1 Configure rootfs

Buildroot

1. Open the configuration menu in your buildroot directory
   ```
   make menuconfig
   ```
2. Activate the sudo package at
   `Target packages -> Shell and utilities -> sudo`
3. Make buildroot and install the new rootfs to the board.

6.4.2 Configure board

1. Make sure you have executed the steps *6.1 Installing VSDebugger to the board* and *6.2 Enabling root access via SSH*
2. Open a Powershell und run the following commands

   ```
   ssh-keygen.exe -m pem

   cat ~/.ssh/id_rsa.pub | ssh root@X.X.X.X "mkdir -p ~/.ssh &&
   cat >> ~/.ssh/authorized_keys"
   ```

   This will create a SSH private and public key and install the public key to your development board. Replace X.X.X.X with your boards IP address.

6.4.3 Configure VS 2019

1. Install VSRemoteDebugger at Extentions > Manage extentions
2. Open Tools > Options > VSRemoteDebugger
3. At *Local Machine Settings* set *Publish* to *True*
4. At *Remote Machine Settings* use the following settings

.NetPath	/usr/share/dotnet/dotnet
Group Name	root
IP Address	Your.board.IP.address
Project folder	/tmp/dotnet
Username	root
Visual Studio Debugger Path	/usr/share/dotnet/vsdbg-linux-arm/vsdbg

- Change *Project folder to* e.g. /opt if you don't want to debug from RAM. Make sure your system is mounted read-writeable.
- Change Visual Studio Debugger Path if you did not install the debugger to the rootfs.

5. Set a breakpoint and run Tools > Start Remote Debugging. You application should get build, transferred to the board and started for debugging.

6.5 Visual Studio 2022

There are two options for remote debugging with Visual Studio 2022. The method described in *6.5.2 Attach to Process* is the "official" way described by Microsoft:

https://learn.microsoft.com/en-us/visualstudio/debugger/remote-debugging-dotnet-core-linux-with-ssh?view=vs-2022

With the extension VSRemoteDebugger the process is simplified, but we cannot guarantee that the description in *6.5.3 VSRemoteDebugger* will work with future versions of Visual Studio or VSRemoteDebugger.

For both methods, SSH on your Linux board must be enabled, see *6.5.1 Configure Board*.

6.5.1 Configure Board

Follow the steps described in *6.4.2 Configure board* to enable SSH connection

6.5.2 Attach To Process

For a more detailed description follow this link:

https://learn.microsoft.com/en-us/visualstudio/debugger/remote-debugging-dotnet-core-linux-with-ssh?view=vs-2022

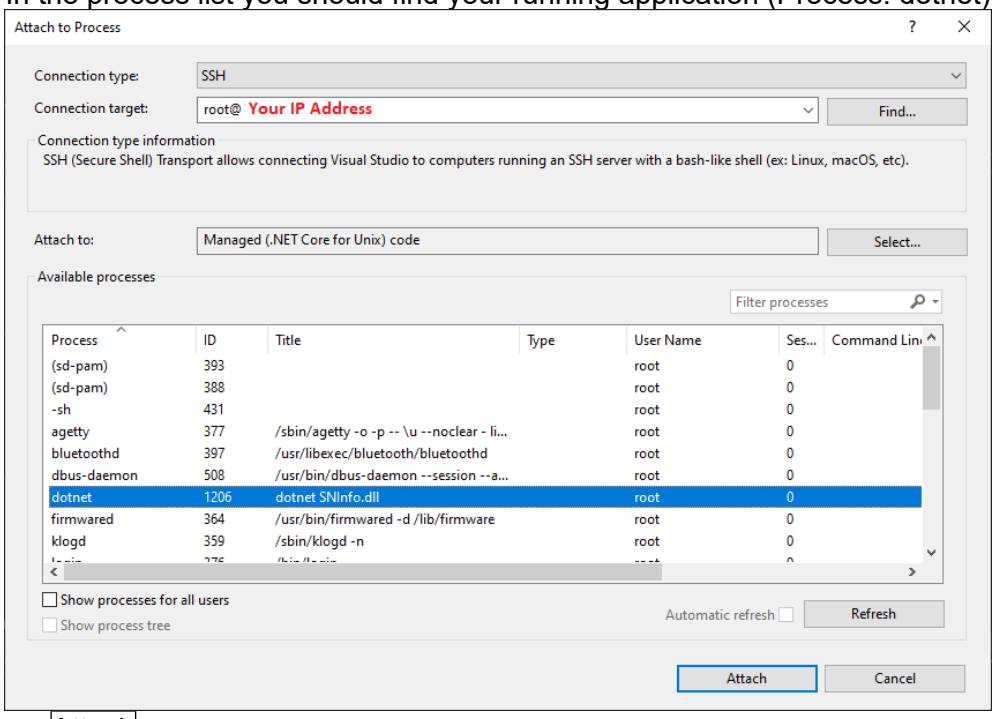
1. Deploy your application for your destination architecture
2. Transfer your build files to the board. You can for example simply use an USB-stick or a tool like WinSCP to establish an SFTP connection to your board. Copy the files to your board.
3. There is also the possibility to use a PowerShell script, which automates the process, see *7.2 Automate the copy process to the board*
4. After transferring the files to your board, execute the .NET applications DLL using Powershell on Windows

```
ssh user@BoardIP

dotnet /path/to/your/application.dll
```

5. When the application is running on your board you can attach to it with the debugger in Visual Studio: Debug > Attach to Process
6. Select Connection Type > SSH
7. Change the Connection Target to the IP address of your board. You will be prompted for your credentials on first connection. Select your private key file, created in *6.5.1 Configure Board*.

8. In the process list you should find your running application (Process: dotnet)



9. Choose Attach
10. You can now start debugging.

7 Advanced Features for Remote Debugging

7.1 Enabling Remote Desktop on Linux

With RDP (Remote Desktop Protocol) you can control your .NET-app from the development machine while it is running on a Linux-Board, without the need of a physical display connected to the board.

7.1.1 Generate keys

First you need to generate keys in Linux. This is only needed to be done once for each board. The first step is to create the directory to store the keys:

```
mkdir /etc/freerdp && mkdir /etc/freerdp/keys/
cd /etc/freerdp/keys/
```

These commands will create the keys:

```
openssl genrsa -out tls.key 2048
openssl req -new -key tls.key -out tls.csr
openssl x509 -req -days 365 -signkey tls.key -in tls.csr -out
tls.crt
```

7.1.2 Start Display Server

The display server won't start automatically if no physical display is connected to your board. Without the display server running, RDP is not working. This can be changed by editing weston.ini:

```
vi /etc/xdg/weston/weston.ini
```

Find the entry for [screen-share]:

```
[screen-share]
command=/usr/bin/weston --backend=rdp-backend.so --shell=fullscreen-shell.so
#start-on-startup=true
```

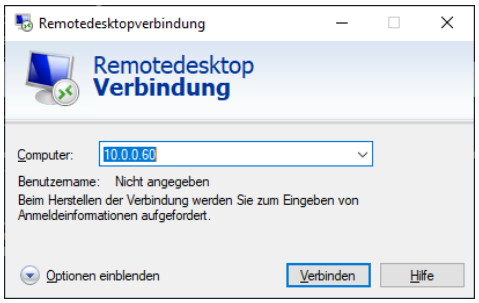
The line '#start-on-startup=true' is commented out. Remove the '#' and save weston.ini to enable the start of the display server on the next boot.

7.1.3 Start RDP Server

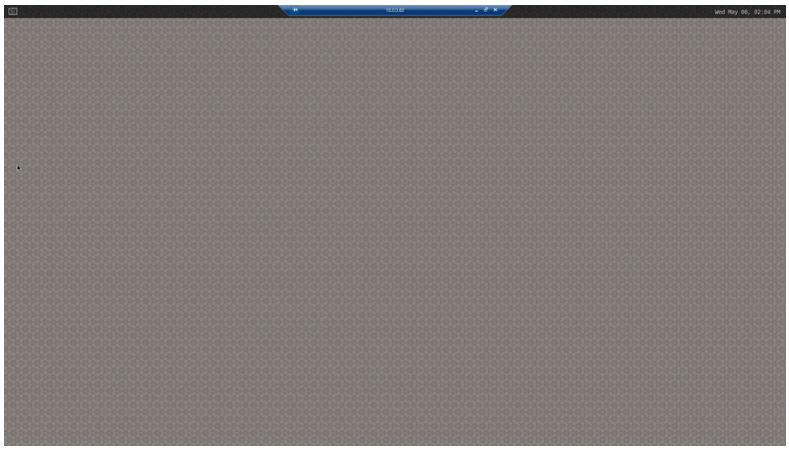
To start RDP on your Linux board, enter this command:

```
/usr/bin/weston --backend=rdp-backend.so --rdp-tls-
cert=/etc/freerdp/keys/tls.crt --rdp-tls-
key=/etc/freerdp/keys/tls.key
```

Now everything is prepared to connect from your Windows host. Start 'Remotedesktop' on Windows and enter the IP-address of your Linux board.



Click connect, Windows may warn because of unverified certificates. If you connect anyway you should see the desktop of your Linux board.



7.1.4 Start App using RDP

If you have no physical display connected to your board, using only RDP, you can start the app as usual, see *5. Executing .NET applications on F&S boards.*

If you have a physical display connected and RDP enabled, you must define on which display the app should run, default is the physical display.

To start the app on RDP, use this command:

```
WAYLAND_DISPLAY=wayland-1 DISPLAY=:1 dotnet
path/to/your/application.dll
```

7.2 Automate the copy process to the board

Instead of manually copying all the generated binaries and libraries needed for execution of your application to the Linux board, you can simply automate this process with a PowerShell script.

You can find an example script at *8.1 copy_debug_to_board.ps1*

This script will copy the content of `.\bin\Debug\net8.0` to a temporary directory, pack this temporary directory as a .tar archive, copy the archive file to your board and unpack it there. It will only copy the relevant runtimes to your board to reduce storage space and make the copy process faster.

Adapt `$ipAddress` and `$projectName` to your conditions. `$runtimesToCopy` can be adapted as well. For Linux you will always need the "*unix*"-runtime, but depending on your board you will only need either "*linux-arm*" or "*linux-arm64*".

Save the PowerShell script to the root directory of your project and let it run after each build to always have the newest binaries on your board.

If you use Visual Studio you can also automate the execution of this script by adding an entry to `[YOUR PROJECT NAME].csproj`:

```
<Target Name="PostBuild" AfterTargets="PostBuildEvent">
    <Exec Command="powershell.exe .\copy_debug_to_board.ps1" />
</Target>
```

If enabled, the files in your Debug directory will automatically be copied to the board whenever you create a new build!

8 Appendix

8.1 copy_debug_to_board.ps1

```
# Define variables, adapt as needed
$ipAddress = "[YOUR BOARD IP ADDRESS]"
$projectName = "[YOUR PROJECT NAME]"

# Directories to copy from the local runtimes directory
$runtimesToCopy = @("unix", "linux-arm", "linux-arm64")
# Local folder where the binaries are stored
$sourceDir = ".\bin\Debug\net8.0"
# Remote board
$remoteHost = "root@${ipAddress}"
$remoteDir = "/home/root"
# Temporary files and folders for the tar process
$tempDir = ".\bin\Debug\tempDir"
$tempSource = "${tempDir}\${projectName}"
$tarFileName = "${projectName}_Debug.tar"
$tarFilePath = ".\bin\Debug\${tarFileName}"

# Function to copy files and folders
function Copy-Files {
    param (
        [string]$source,
        [string]$destination,
        [string]$exclude
    )
    robocopy $source $destination /e /xd $exclude | Out-Null
}

# Copy files and folders to temporary directory, exclude runtimes
directory
Copy-Files -source $sourceDir -destination $tempSource -exclude
"runtimes"

# Copy required runtimes to the temporary directory
foreach ($runtime in $runtimesToCopy) {
```

```
    $runtimePath = Join-Path -Path $sourceDir -ChildPath
"runtimes\$runtime"
    if (Test-Path $runtimePath) {
        Copy-Files -source $runtimePath -destination
"${tempSource}\runtimes\$runtime"
    }
}

# Create .tar archive with all contents from tempDir
tar -cf $tarFilePath -C $tempDir .

# Copy tar archive to remote board
scp $tarFilePath "${remoteHost}:${remoteDir}"
# Extract tar archive on remote board and remove it
ssh $remoteHost "tar -xf ${tarFileName} && rm ${tarFileName}"

# Clean up temporary directory
Remove-Item -Path $tempDir -Recurse
# Delete local .tar file
Remove-Item -Path $tarFilePath
```

8.2 Important Notice

The information in this publication has been carefully checked and is believed to be entirely accurate at the time of publication. F&S Elektronik Systeme GmbH (“F&S”) assumes no responsibility, however, for possible errors or omissions, or for any consequences resulting from the use of the information contained in this documentation.

F&S reserves the right to make changes in its products or product specifications or product documentation with the intent to improve function or design at any time and without notice and is not required to update this documentation to reflect such changes.

F&S makes no warranty or guarantee regarding the suitability of its products for any particular purpose, nor does F&S assume any liability arising out of the documentation or use of any product and specifically disclaims any and all liability, including without limitation any consequential or incidental damages.

Products are not designed, intended, or authorized for use as components in systems intended for applications intended to support or sustain life, or for any other application in which the failure of the product from F&S could create a situation where personal injury or death may occur. Should the Buyer purchase or use a F&S product for any such unintended or unauthorized application, the Buyer shall indemnify and hold F&S and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, expenses, and reasonable attorney fees arising out of, either directly or indirectly, any claim of personal injury or death that may be associated with such unintended or unauthorized use, even if such claim alleges that F&S was negligent regarding the design or manufacture of said product.