.NET on F&S Boards

Version 1.1
(2025-12-26)

© F&S Elektronik Systeme GmbH
Untere Waldplätze 23
D-70569 Stuttgart
Germany

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

About This Document

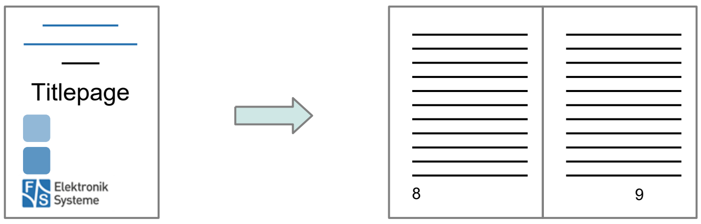
This document describes how to run .NET applications on F&S Boards.

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
2024-05-29	0.9	ALL	M	5.1, 6.3	Changed “Hello World” to .NET8.0 VSCode: new example config in git-repository, changed description and pictures to use new config-files	BS
2025-04-16	1.0	ALL	M	7.1	Optimized the RDP process	BS
2025-12-09	1.1	All	A,M,R	ALL	Remove outdated software (VS2019, .NET <= 7.0), remove Buildroot, new vsdbg process added, add “Known Issues” and more system requirements	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 boards with Yocto Linux.

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

2.1 Storage

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

Yocto

Image type	Image size
Ubifs Image (Nand Flash)	
Ext4 Image (eMMC)	915 MB

If your flash memory is smaller than required for a .NET image, you can still run .NET-applications on your board by publishing your .NET application as a self-contained binary (see chapter *5.2 Compiling and Executing the .NET Application*) or install .NET on an external device (chapter *4.1* and *8.1*).

2.2 Additional Software

For best compatibility, please install the latest F&S release for your device.

For remote debugging with Visual Studio or VS Code, some additional software on your board may be required.

2.2.1 Visual Studio Debugger (vsdbg)

If you want to debug from Visual Studio 2022 or later, you do not need to manually download the debugger. It will be downloaded automatically on your board when you attach to the running .NET applications process. In this case you can skip this chapter.

When using Visual Studio Code or earlier versions of Visual Studio, you can download vsdbg using a script provided by Microsoft (the same script that more modern versions of VS provide automatically as aforementioned).

The GetVsDbg.sh script is available here: https://aka.ms/getvsdbgsh

You can download it directly to your Board with the following command:

```
wget --no-check-certificate https://aka.ms/getvsdbgsh -O
GetVsDbg.sh
```

To download the debugger you need to invoke the script with:

```
chmod +x GetVsDbg.sh

./GetVsDbg.sh -v vs2022 -l .vs-debugger/vs2022
```

`vsdbg` for arm needs about 184 MB of disk space, on arm64 `vsdbg` takes about 206 MB.

If you run out of disk space, see chapter *8.1*.

2.2.2 libicu

Missing ICU libraries may be a problem for .NET on older F&S releases. See chapter *8.2 Missing ICU libraries* for more information.

2.2.3 ps

Older F&S releases have the ‘`ps`’ command contained in `busybox`. There are some options missing from this ‘`ps`’ version which are required by Visual Studios ‘Attach to Process’-Debugger. See chapter *8.3 Missing options in ‘ps’*.

2.2.4 Xwayland

GUI applications with Avalonia need X11 / Xwayland. The images provided by F&S don’t include Xwayland, but instead it must be activated in Yocto.

In short, get the Virtual Machine that fits your board from the F&S download section. The following commands contain `fsimx8mp`, please adapt this to your board.

```
cd fsimx8mp-Y2024.07
./setup-yocto build
cd ./build/yocto-fus
```

Prepare the build, note the ‘xwayland’ in DISTRO:

```
DISTRO=fus-imx-xwayland MACHINE=fsimx8mp . fus-setup-release.sh
```

Finally, build the image:

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

Please have a look at the *Linux on F&S Boards.pdf* documentation, chapter *10.2.3 Xwayland*. Information on our development machines can be found in *Quickstart with FS Development Machine* .For further information please contact the F&S support.

If your image already includes Xwayland, you must activate it in the `weston.ini`:

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

Find the line

```
#xwayland=true
```

and remove the trailing ‘#’. After that, save the file and restart Weston:

```
systemctl restart weston
```

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_36_V1.7
Yocto	fsimx6-Y2024.04.1 fsimx8mp-Y2024.07 fsimx8mm-Y2024.10 fsimx93-Y2025.08
.NET	SDK 10.0.100 .NET Runtime 8.0.22 (on i.MX6, 8, 93) .NET Runtime 10.0.1 (on i.MX8, 93)
vsdbg	18.0.10821.2
Visual Studio Code	1.106.3
Visual Studio Professional 2022	17.14.21
Visual Studio Professional 2026	18.0.2

4 How To Add .NET To Your Board

There are different ways to run .NET applications on your board.

The easiest way for a quick start into development is to include .NET in your published application (self-contained publish). In this case you don't need to modify your existing Linux image. You can skip this chapter and go directly to the next one. Compiling a self-contained application is described in chapter *5.2 Compiling and Executing the .NET Application*.

You could also download a precompiled binary of the .NET runtime directly from Microsoft and install it on your board (chapter *4.2*)

Outside development purposes we recommend to compile a dedicated Linux-Image with included .NET. How to build a root filesystem with preinstalled .NET binaries using Yocto is described in chapter *4.3*.

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

The different ways of how to install the images on an F&S board board are described in the document *Linux on F&S Boards* chapters *Image Download* and *Image Storage*.

Note

Releases based on Yocto 4 (kirkstone) only support up to .NET 8. For more modern .NET versions, please install a newer F&S release if available for your device.

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.

4.1 Installing Precompiled .NET

You can download the .NET binaries from the official Microsoft website: https://dotnet.microsoft.com/en-us/download/dotnet

Make sure to download the right version for 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.

Transfer the downloaded .NET runtime to your board, e.g. by using `scp`. Adapt this command to your needs:

```
scp .\dotnet-runtime-[VERSION]-linux-arm64.tar.gz root@[IP]:~
```

On the board, first create your target directory, then extract the archive into this target:

```
mkdir /PATH/TO/DOTNET

tar -xf [DOTNET ARCHIVE].tar.gz -C /PATH/TO/DOTNET
```

The target directory can also be an external storage device (e.g. an USB stick or SD card, see chapter *8.1* for more expalantions)

Include the .NET directory into your `PATH`:

```
export PATH=/PATH/TO/DOTNET:$PATH
```

To permanently add .NET to your `PATH` you need to add it to `.profile`:

```
echo 'export PATH=/PATH/TO/DOTNET:$PATH' >> ~/.profile
```

4.2 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 existing build:

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

Get the .NET layer 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, change the dotnet-version to your preferred one:

```
PREFERRED_VERSION_dotnet-core = "8.0.1"

CORE_IMAGE_EXTRA_INSTALL += "dotnet-core vsdbg "
```

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. If you already have an existing build, it will only add the dotnet-core and vsdbg to this build:

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

5 Executing .NET applications on F&S boards

This chapter describes how to create a .NET application (console or GUI) on your development machine, compile it for Linux and then transfer and execute it on your board.

This assumes you have a working installation of the .NET SDK on your development machine.

Both application types (CLI / GUI with Avalonia UI) can be compiled for different platforms, which is one of the great benefits of .NET.

5.1 Create a .NET Application

5.1.1 Console Application

To create a basic 'Hello World' application you can use this command, which will create a console application named 'MyApp' based on .NET 8.0:

```
dotnet new console -o MyApp -f net8.0
```

Change your directory to the project directory (where the *.csproj file is located):

```
cd MyApp
```

5.1.2 GUI Application with Avalonia UI

Avalonia is an open source cross-platform UI framework for .NET, with a similar development experience to WPF. It is our recommended UI framework for .NET applications on Linux.

At the moment of writing, Avalonia does net yet support Wayland, but still needs X11 / Xwayland. For more information, see chapter *2.2.4 Xwayland*.

To create a new application named 'MyGuiApp', execute following commands:

```
dotnet new install Avalonia.Templates

dotnet new avalonia.app -o MyGuiApp
```

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.

Change your directory to the project directory (where the `*.csproj` file is located):

```
cd MyGuiApp
```

5.2 Compiling and Executing the .NET Application

This is the basic publish command, which is the same for console and GUI applications:

```
dotnet publish -r [runtime] -c [configuration]
```

Publish your application as `linux-arm` for i.MX6/7 boards and `linux-arm64` for i.MX8 and i.MX9 boards. Please adapt the following commands depending on your hardware and application name.

If you have .NET installed on the board, use the `–no-self-contained` flag to exclude the runtime binaries from your build.

```
dotnet publish -r linux-arm64 -c Debug --no-self-contained
```

If you don’t have .NET installed on the board you can publish your application as self-contained. Add the `--self-contained` flag:

```
dotnet publish -r linux-arm64 -c Debug --self-contained true
```

To transfer your application files to your board you can use `scp`. Adapt the command to your actual platform (linux-arm / linux-arm64), directories and IP address:

```
scp -r .\bin\Debug\net8.0\linux-arm64\publish root@[BOARD
IP]:~/MyApp
```

If you can’t use `scp`, there are other ways to copy the files to your board, as using `WinSCP`, `tftp` or an USB stick. For descriptions see *Linux on F&S Boards* chapter *Using the Standard System and Devices*. If you want to automate the transfer, see *7.2 Automate the copy process to the board*. We use a PowerShell script which is called directly from the `*.csproj` file.

With installed .NET on you board, execute the application by calling its `.dll`:

```
dotnet MyApp/MyApp.dll
```

A self-contained application first needs to be set executable:

```
chmod +x ./MyApp/MyApp
```

After that, you can execute the self-contained .NET application like any other application on Linux:

```
./MyApp/MyApp
```

6 Remote debugging .NET apps on F&S Boards

You can use Visual Studio or 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.

Make sure that you meet the system requirements and have the additional software needed for remote debugging installed (especially `vsdbg` when using VS Code). This is described in chapter *2*. If you face issues in remote debugging, there may be already a solution listed in chapter *8*.

6.1 Visual Studio Code

Visual Studio Code has the great advantage that it is available for multiple platforms. The following instructions work the same for Windows or Linux development hosts.

Please have a look at our GitHub-repository:

https://github.com/FSEmbedded/fs_vscode_dotnet_app_development

This repository contains a working configuration for remote debugging CLI and GUI applications, whether they are framework dependent or self-contained. This configuration also works on Windows or Linux development hosts.

6.1.1 Configuring VSCode

Install the C# Dev Kit Extension to VSCode to enable debugging of C# code:

```
code --install-extension ms-dotnettools.csharp
```

In order to launch the VSDebugger on the board, you will have to create or edit the `launch.json` file. As an example, check out the aforementioned repository on GitHub.

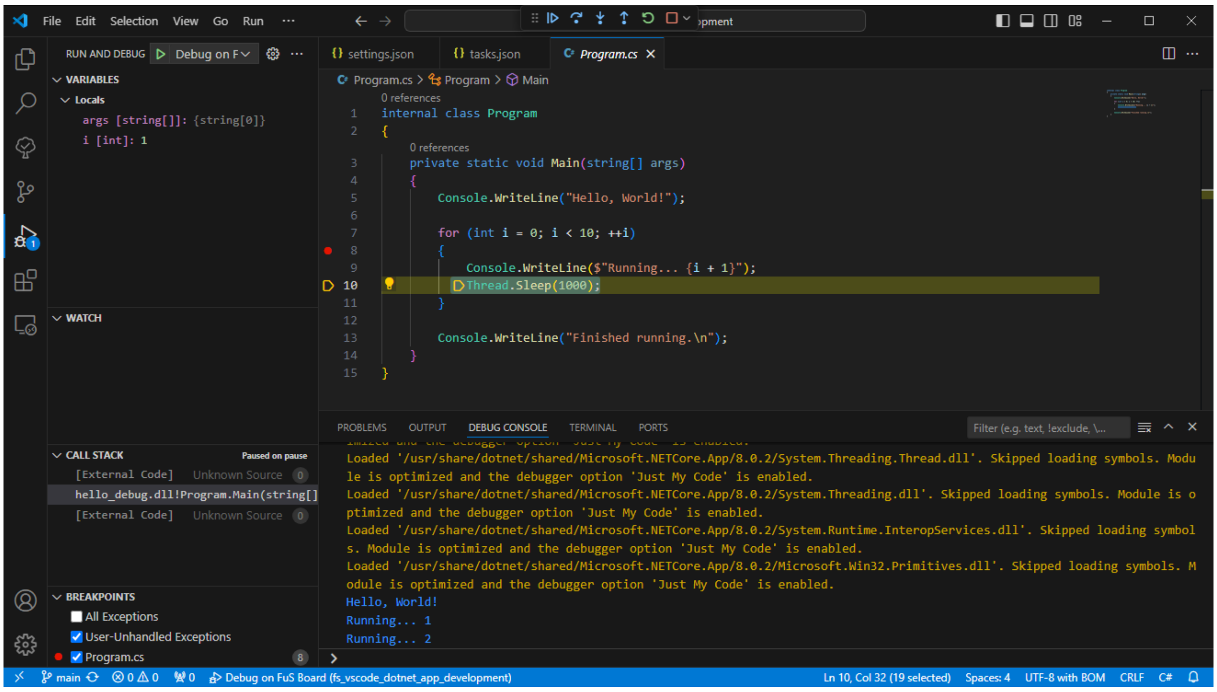
You can also copy the complete '`.vs`'-directory from this repository into your own project directory, adapt the `settings.json` to your needs and start debugging. For explanations of the `settings.json` and more, please refer to the included `README`.

6.1.2 Start debugging in VSCode

Go to the `Run and Debug` tab and select `Debug on FuS Board (.NET)` for framework dependent applications, or `Debug on FuS Board (self-contained)` for self-contained applications.

Click `Start Debugging` or use the shortcut F5 to start the debug process. Your .NET application should be built, deployed to the board and start running. The program will stop at the entry of the Main method and at your set breakpoints.

You can see the output of command line applications in the *Debug Console* inside VS Code.



```
args [string[]]: {string[0]}
i [int]: 1
```

```
[External Code]    Unknown Source
hello_debug.dll!Program.Main(string[]
[External Code]    Unknown Source
```

```csharp
internal class Program
{
    private static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");

        for (int i = 0; i < 10; ++i)
        {
            Console.WriteLine($"Running... {i + 1}");
            Thread.Sleep(1000);
        }

        Console.WriteLine("Finished running.\n");
    }
}
```

```
Loaded '/usr/share/dotnet/shared/Microsoft.NETCore.App/8.0.2/System.Threading.Thread.dll'. Skipped loading symbols. Modu
le is optimized and the debugger option 'Just My Code' is enabled.
Loaded '/usr/share/dotnet/shared/Microsoft.NETCore.App/8.0.2/System.Threading.dll'. Skipped loading symbols. Module is o
ptimized and the debugger option 'Just My Code' is enabled.
Loaded '/usr/share/dotnet/shared/Microsoft.NETCore.App/8.0.2/System.Runtime.InteropServices.dll'. Skipped loading symbol
s. Module is optimized and the debugger option 'Just My Code' is enabled.
Loaded '/usr/share/dotnet/shared/Microsoft.NETCore.App/8.0.2/Microsoft.Win32.Primitives.dll'. Skipped loading symbols. M
odule is optimized and the debugger option 'Just My Code' is enabled.
Hello, World!
Running... 1
Running... 2
```

GUI applications will show up on the display defined in the `settings.json`. This can also be a virtual display using RDP, see chapter 7.1 for more informations.

6.2 Visual Studio

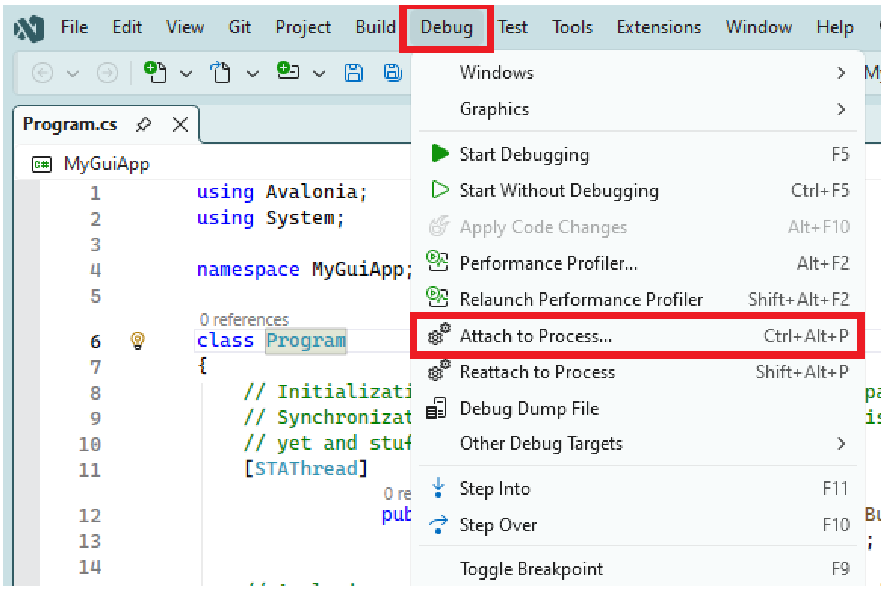
Remote debugging on Linux using SSH works the same way on Visual Studio 2022 and Visual Studio 2026. We mean both of these versions from now on when talking about Visual Studio, without explicitly telling the version.

Older Visual Studio releases might require other configurations that are not included in this guide. Please refer to the official Microsoft article if you are using any other version than 2022 or 2026: https://learn.microsoft.com/en-us/visualstudio/debugger/remote-debugging-dotnet-core-linux-with-ssh?view=visualstudio

6.2.1 Attach To Process

Visual Studios “Attach to Process”, as the name says, attaches to an already running process on your board. So before you can start debugging, you first need to build your application for the destination architecture, then transfer the build files to the board and start the execution. This is already explained in *5.2 Compiling and Executing the Application*.

1. When the application is running on your board you can attach to it with the debugger in Visual Studio: *Debug* > *Attach to Process*

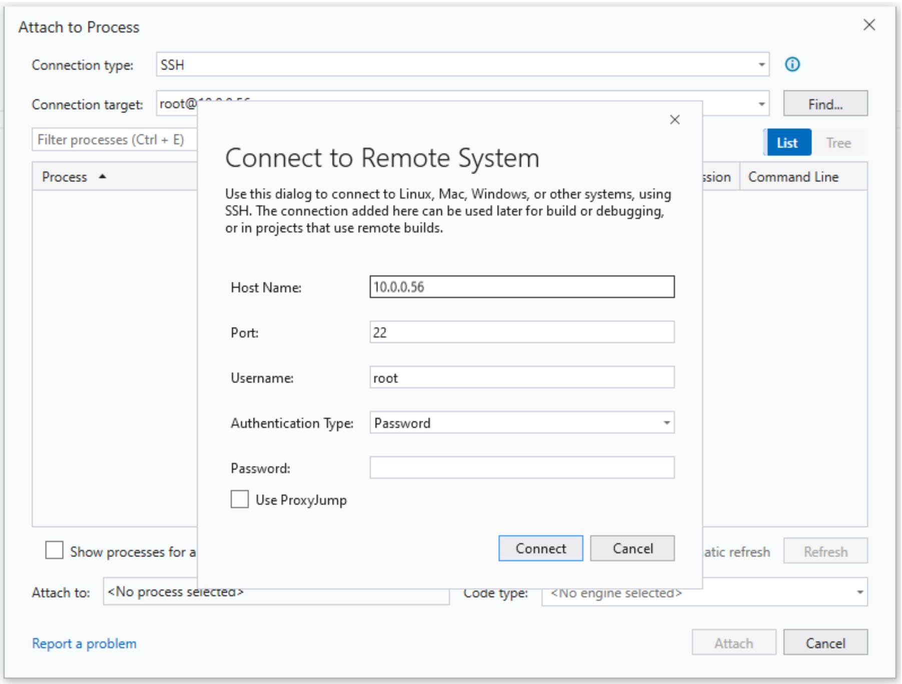


```
using Avalonia;
using System;

namespace MyGuiApp;

0 references
class Program
{
    // Initializati
    // Synchronizat
    // yet and stuf
    [STAThread]
                0 re
                pub
```

2. Select *Connection Type > SSH*
3. 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 or authenticate with password.



Attach to Process

Connection type:	SSH
Connection target:	root@

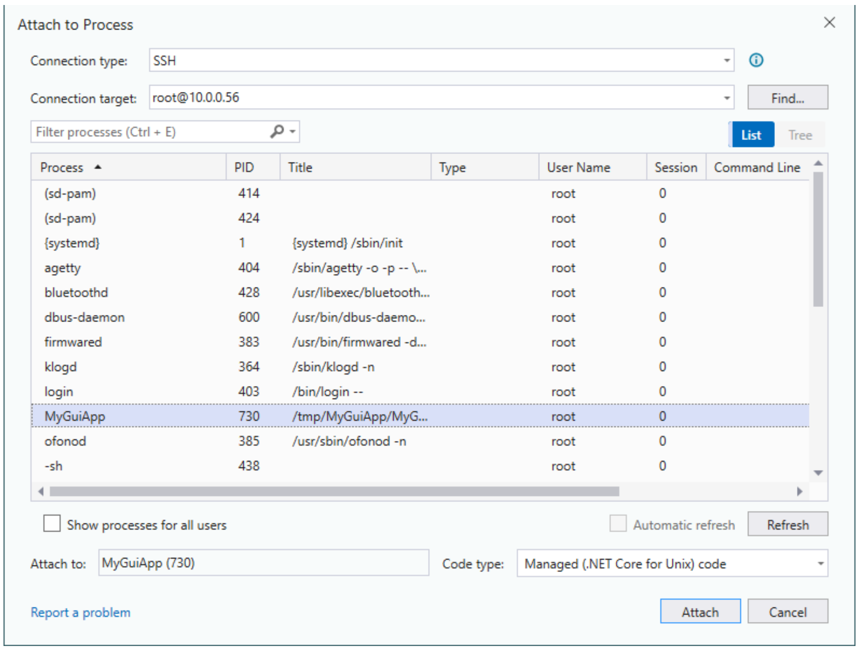
Connect to Remote System

Use this dialog to connect to Linux, Mac, Windows, or other systems, using SSH. The connection added here can be used later for build or debugging, or in projects that use remote builds.

Host Name:	10.0.0.56
Port:	22
Username:	root
Authentication Type:	Password
Password:	

Attach to: <No process selected>
Code type: <No engine selected>

4. In the process list you should find your running application (process `dotnet` for framework dependent applications or the name of your application for self-contained apps).


Attach to Process

Connection type: SSH

Connection target: root@10.0.0.56

Process	PID	Title	Type	User Name	Session	Command Line
(sd-pam)	414			root	0	
(sd-pam)	424			root	0	
(systemd)	1	(systemd) /sbin/init		root	0	
agetty	404	/sbin/agetty -o -p -- \...		root	0	
bluetoothd	428	/usr/libexec/bluetooth...		root	0	
dbus-daemon	600	/usr/bin/dbus-daemo...		root	0	
firmwared	383	/usr/bin/firmwared -d...		root	0	
klogd	364	/sbin/klogd -n		root	0	
login	403	/bin/login --		root	0	
MyGuiApp	730	/tmp/MyGuiApp/MyG...		root	0	
ofonod	385	/usr/sbin/ofonod -n		root	0	
-sh	438			root	0	

Show processes for all users

Attach to: MyGuiApp (730)

Code type: Managed (.NET Core for Unix) code

5. Select *Code type: -> Managed (.NET Core for Unix) code*
6. Choose *Attach*
7. 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 interact with the GUI of your .NET-app from the development machine, while the app is running on a Linux-Board, without the need of a physical display connected to the board.

For F&S releases based on Yocto 5 you can use the application '`DeviceSpy`', available on the F&S website's download section. '`DeviceSpy`' will detect your board when connected to the network and enables you to automatically establish an RDP connection.

For older releases follow these steps to activate RDP.

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 -p /etc/freerdp/keys

cd /etc/freerdp/keys/
```

These commands will create the keys:

```
openssl genrsa -out server.key 2048;

openssl req -new -key server.key -out server.csr;

openssl x509 -req -days 365 -signkey server.key -in server.csr -
out server.crt;
```

7.1.2 Activate Screen Share

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 --no-clients-resize --rdp-tls-
cert=/etc/freerdp/keys/server.crt --rdp-tls-
key=/etc/freerdp/keys/server.key

#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 or Weston restart, even without a physical display attached.

To restart Weston, enter this command:

```
systemctl restart weston
```

You may face problems trying to start RDP as long as no physical display is connected to the board. In this case, create the `override.conf`:

```
mkdir -p /etc/systemd/system/weston.service.d

vi /etc/systemd/system/weston.service.d/override.conf
```

Add this content to the `override.conf`:

```
[Service]

ExecStart=

ExecStart=/usr/bin/weston --backend=rdp-backend.so --
modules=systemd-notify.so --rdp-tls-
cert=/etc/freerdp/keys/server.crt --rdp-tls-
key=/etc/freerdp/keys/server.key
```

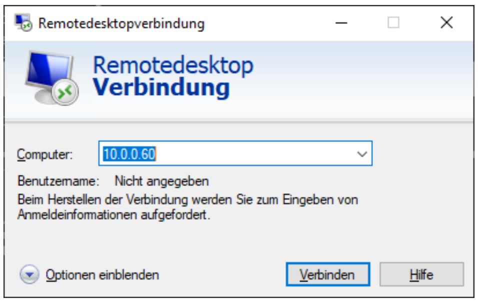
Finally, these commands will activate RDP:

```
systemctl daemon-reload

systemctl restart weston
```

7.1.3 Connect to RDP Server

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


Remotedesktopverbindung

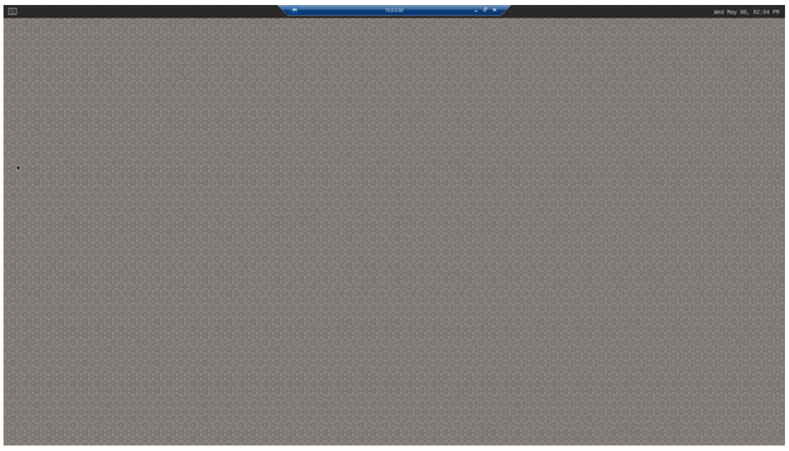
Remotedesktop
Verbindung

Computer: 10.0.0.60

Benutzername: Nicht angegeben

Beim Herstellen der Verbindung werden Sie zum Eingeben von Anmeldeinformationen aufgefordert.

Click connect. Windows may warn you because of unverified certificates. If you connect anyways, you will 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 which display should be used by Weston. Default is the physical display.

To use RDP as the target for the graphical output of your terminal, you must enter this once after each reboot:

```
export DISPLAY=:1
```

If you always want to use RDP as display, you can set this permanently:

```
echo 'export DISPLAY=:1' >> ~/.profile
```

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 Known Issues

You may face some issues, especially while remote debugging. Unfortunately, Visual Studio does not show you the real issue that prevents it from debugging, but instead gives a generic error message. If you can't debug from Visual Studio, you could use VS Code. This will at least give you more feedback of the underlying problems. Fixing them should even solve the issue for Visual Studio.

Following are some common issues we faced in the past, as well as the solutions. Check if you are affected by one or more of them, step by step. All of them will lead to the same error message in Visual Studio.

8.1 Disk Space Not Sufficient

As Visual Studio tries to install the debugger (`vsdbg`) into your user's home directory, it will fail if the partition containing your home directory has not enough free space left. You cannot change the target directory in Visual Studio (as opposed to VS Code). You can see the available space with this command:

```
df -h
```

Check the remaining space for either `/home/root` or `/rootfs`

You can expand your storage with an external device like an USB stick or SD card and place the Visual Studio Debugger, the .NET installation and your application on this device.

Find the partition on your connected storage device:

```
fdisk -l
```

Create a mount point and mount the partition. In this example the mount point is named '`usb`' and the partition is '`/dev/sda1`'. The commands must be adapted to your needs:

```
mkdir /media/usb
mount /dev/sda1 /media/usb
```

If you want this partition to be automatically mounted after a reboot, add it to `/etc/fstab`.

You can now create a link for the VS Debugger from `/home/root` to the external device:

```
mkdir /media/usb/.vs-debugger
ln -s /media/usb/.vs-debugger ~/.vs-debugger
```

To also place the .NET installation on the external storage, have a look at chapter *4.2 Installing Precompiled .NET*.

8.2 Missing ICU libraries

You may face an error like "Couldn't find a valid ICU package installed on the system." when trying to run a .NET application on your board.

If the ICU libraries are missing, this will even lead to an error in VS Debugger, but Visual Studio just gives you a generic error message. So if remote debugging from Visual Studio fails, this may be the underlying problem.

To see if you are affected, check which ICU libraries are installed in your Linux. .NET needs the libraries 'libicuuc.so' and 'libicui18n.so', see if they are listed when running this command:

```
ls /usr/lib/libicu*
```

This issue is solved in newer F&S releases, so if possible please update to the latest version provided on our website.

For older releases, the solution is to add ‘icu’ to your Linux image. In Yocto you can achieve this by adding the following line to the ‘local.conf’ file and then run the bitbake-command:

```
IMAGE_INSTALL:append = " icu "
```

For more information, please see the document *Linux on F&S Boards* chapter *Compiling the System Software* or contact the F&S support.

A quick temporary fix to run your application is to set the globalization invariant mode. This mode enables you to remove application dependencies on globalization data and behavior, but it will lead to poor globalization support. So keep that in mind and see this solution, as said above, only as a quick fix for developing. You activate this mode by setting this environment variable:

```
export DOTNET_SYSTEM_GLOBALIZATION_INVARIANT=1
```

After that, try to start your app again. This variable is only set in your current shell and will reset after a reboot. A permanent fix is to set this mode for every new shell:

```
echo 'export DOTNET_SYSTEM_GLOBALIZATION_INVARIANT=1' >>
~/.profile

source ~/.profile
```

For remote debugging, this variable needs to be also set for SSH-shells:

```
mkdir ~/.ssh

echo 'DOTNET_SYSTEM_GLOBALIZATION_INVARIANT=1' >>
~/.ssh/environment
```

Edit the sshd_config:

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

Edit the following line:

```
#PermitUserEnvironment no      -> PermitUserEnvironment yes
```

After a reboot, remote debugging should work.

You could also add globalization packages directly into your application, then the globalization invariant mode on Linux must only be set for remote debugging / SSH.

Add these lines to the **.csproj* file of your application, you might change the versions of the added packages:

```
<PropertyGroup>

  <InvariantGlobalization>false</InvariantGlobalization>

</PropertyGroup>

<ItemGroup>
```

```
  <RuntimeHostConfigurationOption
Include="System.Globalization.AppLocalIcu" Value="72.1.0.3" />

  <PackageReference Include="Microsoft.ICU.ICU4C.Runtime"
Version="72.1.0.3" />

</ItemGroup>
```

8.3 Missing options in ‘ps’

For remote debugging from Visual Studio, a working installation of ‘ps’ with enhanced options is required on your board. This is provided in releases with Yocto 5.15 and onwards.

You can test your installed ‘ps’ by executing the following command in Linux on your board:

```
ps axww
```

This command is used by Visual Studios “Attach to Process” to list all running processes. If this command gives you a list of running processes, you can skip the following. However, the command may also give you an error like this:

```
ps: invalid option -- 'a'

BusyBox v1.36.1 (2025-10-07 15:42:51 CEST) multi-call binary.
```

Remote Debugging from Visual Studio will fail in this case.

If possible, update to the latest release provided by F&S. As said, a working ‘ps’ is included in newer images. If you want to install a standalone ps to your own Yocto image, add the following line to the local.conf, then run the bitbake-command:

```
IMAGE_INSTALL:append = " procps "
```

For further explanations, please see the document *Linux on F&S Boards* chapter *Compiling the System Software* or ask the F&S support.

8.4. Missing libX11

Avalonia needs X11 / Xwayland to run. Please refer to chapter *2.2.4 Xwayland*.

9 Appendix

9.1 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
```

9.2 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.