Technische Hochschule
Brandenburg
University of
Applied Sciences
Fachbereich
Wirtschaft

SpeedCameraPi

# Developer Manuals

v1.3.0

last updated: October 24, 2023

# Contents

# Part I

# Application Compiling Guide

# 1 Acquiring the Source Code

There are two ways to acquire the source code for the project. They are:

1. Downloading the source code from the GitHub repository
2. Cloning the GitHub repository

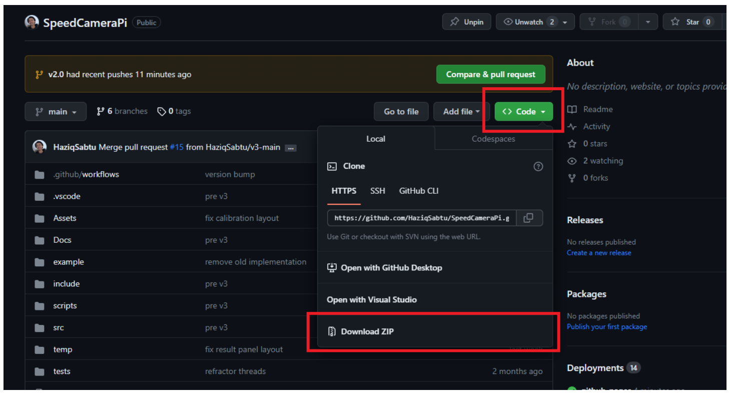
## 1.1 Downloading the Source Code

Alternatively, you can download the source code as a zip file from the GitHub repository page and extract it to your home directory.

- Go to the GitHub repository page:

```
    https://github.com/HaziqSabtu/SpeedCameraPi.git
```

- Click on the green button labeled **Code** and select **Download ZIP**.



| | | |
|---|---|---|
| .github/workflows | version bump | |
| .vscode | pre v3 | |
| Assets | fix calibration layout | |
| Docs | pre v3 | |
| example | remove old implementation | |
| include | pre v3 | |
| scripts | pre v3 | |
| src | pre v3 | |
| temp | fix result panel layout | |
| tests | refactor threads | 2 months ago |

- Extract the downloaded zip file to your **home** directory.

## 1.2 Cloning the GitHub Repository

Run the following commands to clone the repository:

```
cd ~
git clone https://github.com/HaziqSabtu/SpeedCameraPi.git
```

# 2 Installing Required Dependencies

In this chapter, we will be installing the required dependencies for the project. The dependencies are:

1. OpenCV v4.5.5
2. wxWidgets
3. libcamera v0.0.4

## 2.1 Preparing the system

Before installing the required dependencies, following steps are required to be done:

- Navigate to scripts directory:

```
    cd ~/SpeedCameraPi/scripts/Installation
```

- Allow script to be executed:

```
    sudo chmod +x DI_Scripts1.sh
```

- Run script:

```
    sudo ./DI_Scripts1.sh
```

## 2.2 Preparing the system

Before installing the required dependencies, following steps are required to be done:

- Navigate to scripts directory:

```
    cd ~/SpeedCameraPi/scripts/Installation
```

- Allow script to be executed:

```
    sudo chmod +x DI_Scripts1.sh
```

- Run script:

```
    sudo ./DI_Scripts1.sh
```

### 2.2.1 Explanation of script

- Update the package list and upgrade all packages to their latest versions:

```
    sudo apt-get update && sudo apt-get upgrade
```

- Increase swap size to 4096MByte to prevent the system from running out of memory during the compilation process:

```
    NEW_SWAPSIZE=4096
    sudo sed -i "s/CONF_SWAPSIZE=.*/CONF_SWAPSIZE=$NEW_SWAPSIZE/
       " /etc/dphys-swapfile
    sudo sed -i "s/CONF_MAXSWAP=.*/CONF_MAXSWAP=$NEW_SWAPSIZE/"
       /sbin/dphys-swapfile
    sudo systemctl restart dphys-swapfile
```

## 2.3 Installing OpenCV

OpenCV is a powerful computer vision library that provides a vast range of image and video processing capabilities. This project uses OpenCV to perform image processing tasks such as detecting and tracking objects in the video stream.

Steps to install OpenCV on Raspberry Pi 4 running Debian: [source]

- Navigate to scripts directory:

```
    cd ~/SpeedCameraPi/scripts/Installation
```

- Allow script to be executed:

```
    sudo chmod +x DI_Scripts2.sh
```

- Run script:

```
    sudo ./DI_Scripts2.sh
```

## 2.4 Explanation of script

- Update the package list and upgrade all packages to their latest versions:

```
    sudo apt-get update && sudo apt-get upgrade
```

- Install the required dependencies for building OpenCV:

```
    sudo apt-get install -y build-essential cmake git unzip pkg-
       config
    sudo apt-get install -y libjpeg-dev libtiff-dev libpng-dev
    sudo apt-get install -y libavcodec-dev libavformat-dev
       libswscale-dev
    sudo apt-get install -y libgtk2.0-dev libcanberra-gtk*
       libgtk-3-dev
    sudo apt-get install -y libgstreamer1.0-dev gstreamer1.0-
       gtk3
```

```
sudo apt-get install -y libgstreamer-plugins-base1.0-dev
   gstreamer1.0-gl
sudo apt-get install -y libxvidcore-dev libx264-dev
sudo apt-get install -y python3-dev python3-numpy python3-
   pip
sudo apt-get install -y libtbb2 libtbb-dev libdc1394-22-dev
sudo apt-get install -y libv4l-dev v4l-utils
sudo apt-get install -y libopenblas-dev libatlas-base-dev
   libblas-dev
sudo apt-get install -y liblapack-dev gfortran libhdf5-dev
sudo apt-get install -y libprotobuf-dev libgoogle-glog-dev
   libgflags-dev
sudo apt-get install -y protobuf-compiler
```

- Download OpenCV version 4.5.5 from the repository:

```
cd ~
sudo rm -rf opencv*
wget -O opencv.zip https://github.com/opencv/opencarchive
   /4.5.5.zip
wget -O opencv_contrib.zip https://github.com/
   opencopencv_contrib/archive/4.5.5.zip
unzip opencv.zip
unzip opencv_contrib.zip
mv opencv-4.5.5 opencv
mv opencv_contrib-4.5.5 opencv_contrib
rm opencv.zip
rm opencv_contrib.zip
```

- Create a build directory and navigate to it:

```
cd ~/opencv
mkdir build
cd build
```

- Configure the build process by running the following command:

```
cmake -D CMAKE_BUILD_TYPE=RELEASE \
-D CMAKE_INSTALL_PREFIX=/usr/local \
-D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules \
-D ENABLE_NEON=ON \
-D WITH_OPENMP=ON \
-D WITH_OPENCL=OFF \
-D BUILD_TIFF=ON \
-D WITH_FFMPEG=ON \
-D WITH_TBB=ON \
-D BUILD_TBB=ON \
-D WITH_GSTREAMER=ON \
-D BUILD_TESTS=OFF \
-D WITH_EIGEN=OFF \
-D WITH_V4L=ON \
-D WITH_LIBV4L=ON \
-D WITH_VTK=OFF \
-D WITH_QT=ON \
-D OPENCV_ENABLE_NONFREE=ON \
-D INSTALL_C_EXAMPLES=OFF \
-D INSTALL_PYTHON_EXAMPLES=OFF \
-D OPENCV_FORCE_LIBATOMIC_COMPILER_CHECK=1 \
-D PYTHON3_PACKAGES_PATH=/usr/lib/python3/dist-packages \
-D OPENCV_GENERATE_PKGCONFIG=ON \
-D BUILD_EXAMPLES=OFF ..
```

- Compile and install the library:

```
    make -j1
    sudo make install
    sudo ldconfig
```

## 2.5 Installing wxWidget

wxWidgets is a cross-platform GUI library that provides a set of C++ classes for creating graphical user interfaces. This project uses wxWidgets to create the graphical user interface for the application.

Steps to install wxWidget on Raspberry Pi 4 running Debian: [source]

- Navigate to scripts directory:

```
    cd ~/SpeedCameraPi/scripts/Installation
```

- Allow script to be executed:

```
sudo chmod +x DI_Scripts3.sh
```

- Run script:

```
sudo ./DI_Scripts3.sh
```

### 2.5.1 Explanation of script

- Update the package list and upgrade all packages to their latest versions:

```
sudo apt-get update && sudo apt-get upgrade
```

- Install the required dependencies for building wxWidgets:

```
sudo apt-get install libgtk-3-dev build-essential
    checkinstall
```

- Download the latest stable version of wxWidgets from the official website:

```
wget https://github.com/wxWidgets/wxWidgets/releases/
    download/v3.2.2.1/wxWidgets-3.2.2.1.tar.bz2
```

- Extract the downloaded archive and navigate to the extracted directory:

```
tar -xvf wxWidgets-3.2.2.1.tar.bz2
```

- Change directory to the extracted directory:

```
cd wxWidgets-3.2.2.1
```

- Create a build directory and navigate to it:

```
mkdir gtk-build
cd gtk-build
```

- Configure the build process by running the following command:

```
../configure --disable-shared --enable-unicode --disable-
    debug
```

- Compile and install the library:

```
    make
    sudo make install
```

### 2.5.2 Installing libcamera

Libcamera is an open-source software stack designed to support complex camera configurations on Raspberry Pi and other embedded devices. It provides a unified interface for multiple camera modules, allowing developers to efficiently harness their full potential in various applications, from computer vision projects to advanced photography.

As of now, libcamera is still under development and is still in alpha stage. However, it is already being used by the Raspberry Pi Foundation to support the Raspberry Pi High-Quality Camera Module. It is recommended to use libcamera in the future as it provides a more flexible and powerful interface for controlling the camera module.

For this project, we will be using v0.0.4. As of now, a higher version is available, however, as it is still in the alpha stage, a lot of changes are still being made. When a more stable release is available, we will update this guide.

Steps to install Libcamera v0.0.4 on Raspberry Pi 4 running Debian:

- Navigate to scripts directory:

```
    cd ~/SpeedCameraPi/scripts/Installation
```

- Allow script to be executed:

```
    sudo chmod +x DI_Scripts4.sh
```

- Run script:

```
    sudo ./DI_Scripts4.sh
```

### 2.5.3 Explanation of script

- Update the package list and upgrade all packages to their latest versions:

```
    sudo apt-get update && sudo apt-get upgrade
```

- Install the required dependencies for building libcamera:

```
    sudo apt install -y libboost-dev
    sudo apt install -y libgnutls28-dev openssl libtiff5-dev
       pybind11-dev
    sudo apt install -y qtbase5-dev libqt5core5a libqt5gui5
       libqt5widgets5
    sudo apt install -y meson
    sudo apt install -y cmake
    sudo pip3 install pyyaml ply
    sudo pip3 install --upgrade meson
    sudo apt install -y libglib2.0-dev libgstreamer-plugins-
       base1.0-dev
```

- Download the repository and checkout to v0.0.4:

```
    cd ~
    git clone https://github.com/raspberrypi/libcamera.git
    cd libcamera
    git checkout v0.0.4
```

- Configure and build the library:

```
    meson build --buildtype=release -Dpipelines=raspberrypi -
       Dipas=raspberrypi -Dv4l2=true -Dgstreamer=enabled -Dtest=
       false -Dlc-compliance=disabled -Dcam=disabled -Dqcam=
       enabled -Ddocumentation=disabled -Dpycamera=enabled
    ninja -C build -j2    # use -j 2 on Raspberry Pi 3 or earlier
        devices
    sudo ninja -C build install
```

### 2.5.4 Clean up

After installing the required dependencies, you may want to clean up the unnecessary files to free up some disk space. To do so, run the following commands:

- Navigate to scripts directory:

```
    cd ~/SpeedCameraPi/scripts/Installation
```

- Allow script to be executed:

```
    sudo chmod +x DI_Scripts5.sh
```

- Run script:

```
    sudo ./DI_Scripts5.sh
```

# 3 Compiling the Application

To build the application, follow these steps:

- Create a build directory and navigate to it:

```
    cd ~/SpeedCameraPi
    mkdir build
    cd build
```

- Configure the build process by running the following command:

```
    cmake -B. -H..
```

- Compile the application:

```
    make -j3
```

- After finished compiling, the executable file is located in the build directory.

```
    ./build
```

- To run the application, simply double-click the executable file or run the following command:

```
    ./SpeedCameraPi
```

# 4 Utility Scripts

This section will provide some useful scripts. These scripts are optional and are not required to run the application. However, they are useful for managing the application. The scripts are:

- **Hotspot.sh** - This script will create a hotspot on the Raspberry Pi. This is useful if you want to connect to the Raspberry Pi without a router.
- **Startup.py** - This script will run the application on startup. This is useful if you want the application to run automatically when the Raspberry Pi boots up.
- **Touchscreen.sh** - This script will fix the Touchscreen rotation on the Raspberry Pi.

## 4.1 Hotspot

This script will configure the Raspberry Pi to enable the hotspot feature whenever it is not connected to the local network. This will allow you to connect to the Raspberry Pi directly when you are not at home.

- Navigate to scripts directory:

```
    cd ~/SpeedCameraPi/scripts/Installation
```

- Allow script to be executed:

```
    sudo chmod +x Hotspot.sh
```

- Run script:

```
    sudo ./Hotspot.sh
```

## 4.2 Startup

This script will configure the Raspberry Pi to automatically start the SpeedCameraPi application whenever it is booted up. This will allow you to use the Raspberry Pi as a standalone device without the need to manually start the application.

- Navigate to scripts directory:

```
    cd ~/SpeedCameraPi/scripts/Installation
```

- Allow script to be executed:

```
    sudo chmod +x Startup.py
```

- Run script:

```
    sudo ./Startup.py
```

## 4.3 Touchscreen

This script will fix the touchscreen rotation issue. This will allow you to use the touchscreen in portrait mode.

- Navigate to scripts directory:

```
    cd ~/SpeedCameraPi/scripts/Installation
```

- Allow script to be executed:

```
    sudo chmod +x Touchscreen.sh
```

- Run script:

```
    sudo ./Touchscreen.sh
```

# Part II

# Setting up the Development Environment

For future developers, this section will help you properly set up your development environment. In general, the development is done with Visual Studio Code with cross-compilation to Raspberry Pi 4. A guide to setting up the development environment is provided below. Alternatively, a native development setup will also be provided.

# 5 Introduction

C++ is a programming language that is designed to be compiled, meaning that it is generally translated into machine language that can be understood directly by the system, making the generated program highly efficient. To compile C++ code, you need a set of tools known as the development toolchain, whose core are a compiler and its linker. A compiler is a program that converts your C++ code into machine code, while a linker is a program that combines the machine code with other libraries and resources to create an executable file. There are different compilers and linkers available for C++, such as Visual Studio, Clang, mingw, etc. Some of them are integrated with IDEs (Integrated Development Environments) that provide features such as syntax highlighting, debugging, code completion, etc.

To cross-compile C++ code, you need to use a cross-compiler that can generate executable code for a different platform than the one you are running on. For example, if you want to compile C++ code for ARM architecture on a Windows PC, you need to use a cross-compiler like arm-none-linux-gnueabi-gcc. Depending on the tool you are using, you may need to configure some settings to enable cross compilation. For example, if you are using Visual Studio Code, you can edit the Compiler path and Compiler arguments settings in the C/C++ extension to specify the cross-compiler and the target triplet. You can also use other tools like CMake or Makefile to automate the cross-compilation process.

# 6 Cross-compilation

Cross-compilation is the process of compiling code on one platform (host) to run on another platform (target). In this case, we will be compiling the code on a Windows 11 machine running Windows Subsystem for Linux (WSL) to run on Raspberry Pi 4 running Raspberry Pi OS (Bullseye 32-bit).

This method of development is preferred as it allows for faster compilation time and easier debugging, which allows for better development experience.

Upon further reading, there are two important terms to take note of:

- **Host** - The machine that you are developing on. In this case, it is the Windows 11 machine.
- **Target** - The machine that you are developing for. In this case, it is the Raspberry Pi 4.

This method has been tested numerous times and is proven to work. However, it is not guaranteed to work on all machines. To reproduce the development environment, following Build number of Windows 11 and WSL is required:

- Windows 11 Pro Version 22H2 OS Build 22621.2428
- WSL 2 with Ubuntu 22.04 LTS

## 6.1 Steps to setup Cross-Compilation

For this section, it is assumed that you have already installed WSL 2 with Ubuntu 22.04 LTS. If not, please refer to the official documentation to install WSL 2 with

Ubuntu 22.04 LTS. Most of the process are taken from this guide.

- To begin with, it is important that required dependencies are installed on **Target**. Refer to the Section 2 to install the required dependencies.
- Next, we need to prepare the **Target** to allow ssh connection from the host. To do so, run the following command on **Host**:

  ```
          cd ~/SpeedCameraPi/scripts/crosscompile
          sudo chmod +x setup_ssh.py
          ./setup_ssh.py hostname username
  ```

  Type yes when prompted. When successful, you should now be able to ssh into the target from the host with the following command:

  ```
          ssh RPi0
  ```

- Next, we need to prepare the **target** for cross-compilation. To do so, run the following command on **Host**:

  ```
          cd ~/SpeedCameraPi/scripts/crosscompile
          sudo chmod +x Target_setup.py
          ./Target_setup.py credential password
  ```

- Next, we need to prepare the **host** for cross-compilation. To do so, run the following command on **Host**:

  ```
          cd ~/SpeedCameraPi/scripts/crosscompile
          sudo chmod +x Host_setup.py
          sudo ./Host_setup.py
  ```

- Lastly, now we need to sync the files structure of **Target** with **Host** on the staging directory (sysroot). To do so run following command:

  ```
          cd ~/SpeedCameraPi/scripts/crosscompile
          sudo chmod +x Host_postSetup.py
          ./Host_postSetup.py
  ```

- Now, you should be able to perform cross-compilation. However, as of now, there is a linker problem with OpenCV libraries. To fix this, please refer to the next section.

## 6.2 Fixing OpenCV linker problem

Now, we will once again compile OpenCV. There seems to be a problem using the already compiled library of OpenCV as done previously on the **Target**. Some *opencv_contrib* libraries cannot be found during runtime. Possible linker error? Still investigating. However, the current fix is to cross compile the library on the **Host** and transfer the compiled libs to the **Target**.

- To do so, follow these steps:

```
        cd ~/SpeedCameraPi/scripts/crosscompile
        sudo chmod +x Target_cvinstall.py
        ./Target_cvinstall.py credential
```

- Now, we will compile OpenCV on **host** machine:

```
        cd ~/SpeedCameraPi/scripts/crosscompile
        sudo chmod +x Host_cvcompile.py
        sudo ./Host_cvcompile.py
```

  In some cases, during compilation of OpenCV, it will produce error *jconfig.h not found*. To fix this, run the following command:

```
        cd ~/SpeedCameraPi/scripts/crosscompile
        sudo chmod +x fix_jconfig.py
        ./fix_jconfig.py
```

- Now, when the library is compiled, we can transfer the compiled library to the **target** machine. To do so, run the following command on **Host**:

```
        cd ~/SpeedCameraPi/scripts/crosscompile
        sudo chmod +x Host_cvinstall.py
        ./Host_cvinstall.py
```

Now, the machine is ready for cross-compilation. Now we will explore steps to install Visual Studio Code.

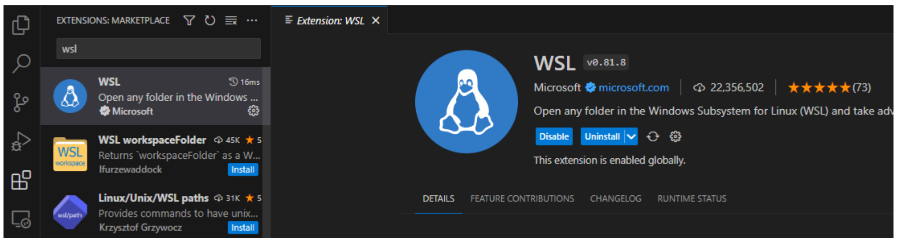
## 6.3 Installing Visual Studio Code

Visual Studio Code is a free source-code editor made by Microsoft for Windows, Linux, and macOS. It is a very popular code editor among developers. This section will guide you through the process of installing Visual Studio Code for cross-compilation.

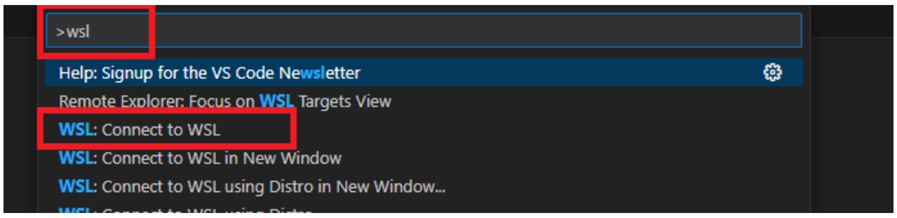
If you are running Linux on WSL, it is recommended to install Visual Studio Code directly on Windows. Download the installer from here and install it on Windows.

## 6.4 Accessing WSL from Visual Studio Code

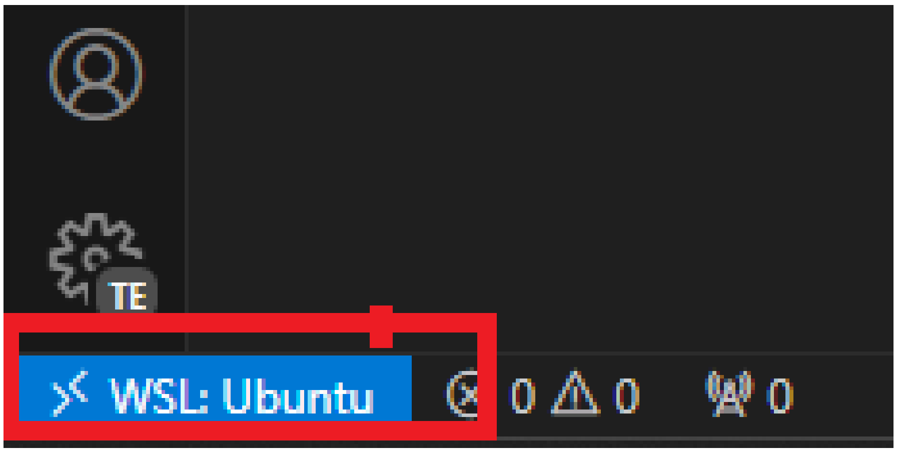
First, we need to install the WSL Extension for Visual Studio Code. To do so, open the Extensions pane (Ctrl+Shift+X) and search for WSL. Click on the Install button to install the extension.


EXTENSIONS: MARKETPLACE
wsl
WSL
Open any folder in the Windows ...
Microsoft
WSL workspaceFolder
Returns `workspaceFolder` as a W...
Ifurzewaddock
Install
Linux/Unix/WSL paths
Provides commands to have unix...
Krzysztof Grzywocz
Install
Extension: WSL
WSL v0.81.8
Microsoft microsoft.com 22,356,502 (73)
Open any folder in the Windows Subsystem for Linux (WSL) and take adv
Disable
Uninstall
This extension is enabled globally.
DETAILS
FEATURE CONTRIBUTIONS
CHANGELOG
RUNTIME STATUS

To access WSL from Visual Studio Code, open the Command Palette (Ctrl+Shift+P) and type `WSL: Connect to WSL`. This will open a new Visual Studio Code window with WSL as the default shell. You can also open a folder in WSL by right-clicking on the folder in the File Explorer and selecting Open in WSL.


>wsl
Help: Signup for the VS Code Newsletter
Remote Explorer: Focus on WSL Targets View
WSL: Connect to WSL
WSL: Connect to WSL in New Window
WSL: Connect to WSL using Distro in New Window...

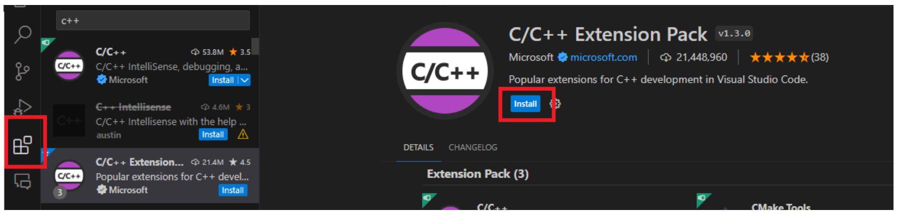
You should see that you are currently in the WSL shell. You can also check the bottom left corner of the window to see if you are in the WSL shell.



## 6.5 Installing Required Extensions

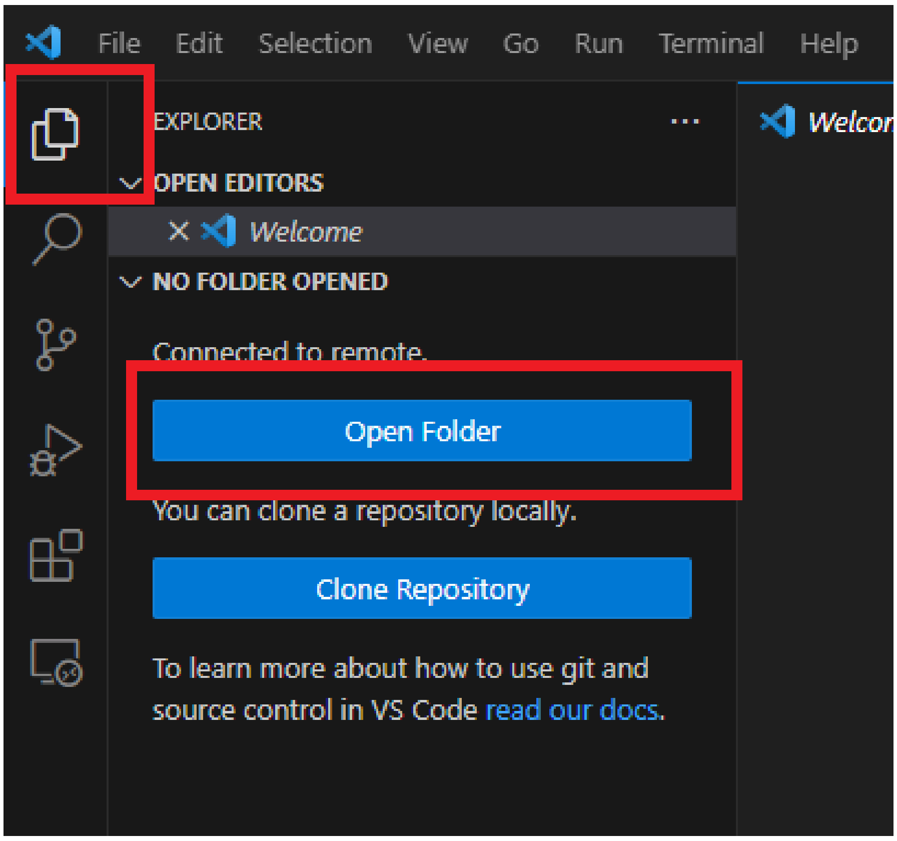
To install extensions, open Visual Studio Code and press Ctrl+Shift+X to open the Extensions pane. Search for the following extensions and install them:

- C/C++ Extension Pack


C/C++ Extension Pack v1.3.0
Microsoft microsoft.com 21,448,960 ★★★★★(38)
Popular extensions for C++ development in Visual Studio Code.
Install
DETAILS
CHANGELOG
Extension Pack (3)

## 6.6 Opening SpeedCameraPi project

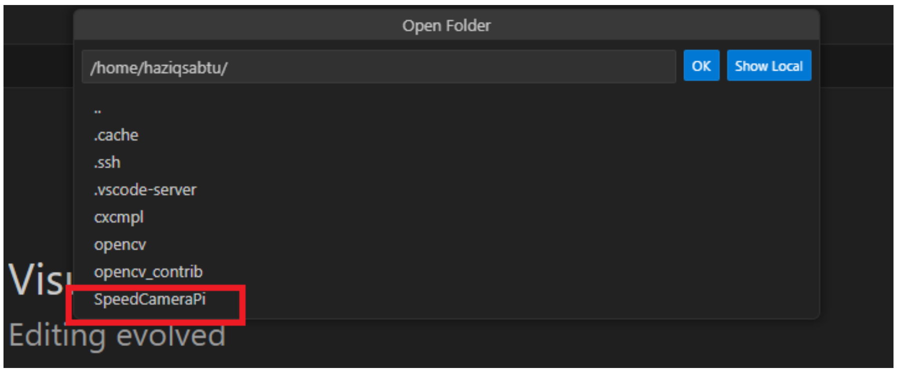
To open the SpeedCameraPi project, open the Explorer pane (Ctrl+Shift+E) and click on the Open Folder button. Navigate to the SpeedCameraPi folder and click on the OK button. You should now see the SpeedCameraPi project in the Explorer pane.



Connected to remote.

You can clone a repository locally.

To learn more about how to use git and source control in VS Code read our docs.



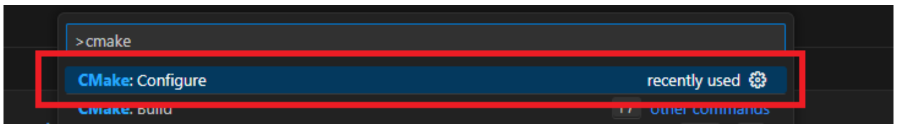
/home/haziqsabtu/

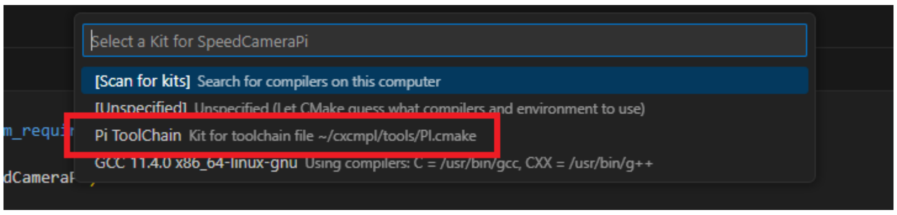
..
.cache
.ssh
.vscode-server
cxcmpl
opencv
opencv_contrib
SpeedCameraPi

## 6.7 Configuring C/C++ Extension

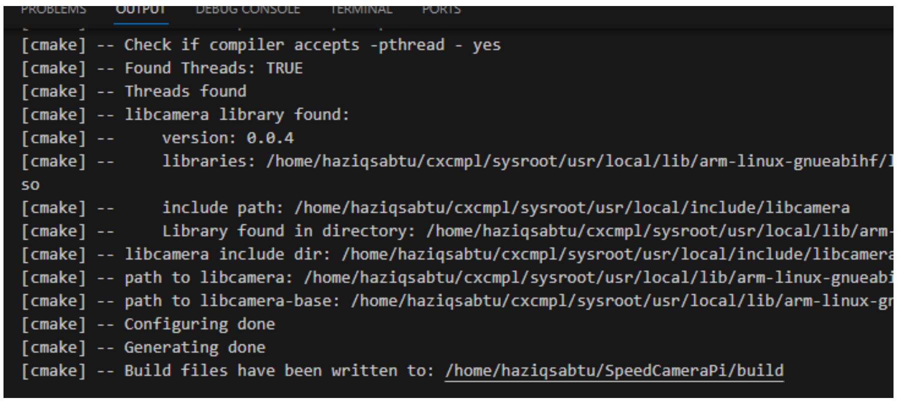
To configure the C/C++ extension, open the Command Palette (Ctrl+Shift+P) and type `CMake:Configure` and select the `Pi Toolchain` option. This will create a `build` folder in the project directory and generate the CMake cache.

If the `Pi Toolchain` option is not available, you may need to check if the cross-compilation setup is done correctly. Additionally, in some cases the path may be different. If so, you may need to edit the `.vscode/cmake-kits.json` file to change the path to the cross compiler.




Select a Kit for SpeedCameraPi
[Scan for kits] Search for compilers on this computer
[Unspecified] Unspecified (Let CMake guess what compilers and environment to use)
Pi ToolChain Kit for toolchain file ~/cxcmpl/tools/PI.cmake
GCC 11.4.0 x86_64-linux-gnu Using compilers: C = /usr/bin/gcc, CXX = /usr/bin/g++

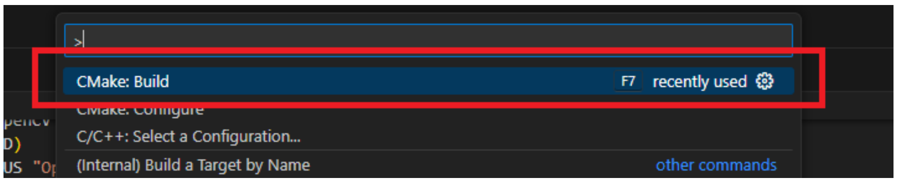
If the configuration process is successful, you should see the following message in the terminal:



```
[cmake] -- Check if compiler accepts -pthread - yes
[cmake] -- Found Threads: TRUE
[cmake] -- Threads found
[cmake] -- libcamera library found:
[cmake] --     version: 0.0.4
[cmake] --     libraries: /home/haziqsabtu/cxcmpl/sysroot/usr/local/lib/arm-linux-gnueabihf/l
so
[cmake] --     include path: /home/haziqsabtu/cxcmpl/sysroot/usr/local/include/libcamera
[cmake] --     Library found in directory: /home/haziqsabtu/cxcmpl/sysroot/usr/local/lib/arm-
[cmake] -- libcamera include dir: /home/haziqsabtu/cxcmpl/sysroot/usr/local/include/libcamera
[cmake] -- path to libcamera: /home/haziqsabtu/cxcmpl/sysroot/usr/local/lib/arm-linux-gnueabi
[cmake] -- path to libcamera-base: /home/haziqsabtu/cxcmpl/sysroot/usr/local/lib/arm-linux-gr
[cmake] -- Configuring done
[cmake] -- Generating done
[cmake] -- Build files have been written to: /home/haziqsabtu/SpeedCameraPi/build
```

## 6.8 Building the project

To build the project, open the Command Palette (Ctrl+Shift+P) and type `CMake:Build`. This will build the project and generate the executable file in the `build` folder.


>|
CMake: Build
F7 recently used
CMake: Configure
C/C++: Select a Configuration...
(Internal) Build a Target by Name
other commands

Within the `CMakeLists.txt` file, the executable file is by default configured to be automatically copied to the **Target** machine. You can disable this by editing out the command. However, it is recommended to keep it as it is. The build application is located at ~/`Target/SpeedCameraPi`.

## 6.9 Debugging the project

### 6.9.1 Preparations

Debugging is crucial when developing a project. These steps need to be done at least once. To perform debugging, we first need to configure the `launch.json` script. To do so, navigate to the script folder:

```
cd ~/SpeedCameraPi/scripts/debug
```

Next, run the following command:

```
sudo chmod +x launch.py
```

Lastly, run the script with the following parameters:

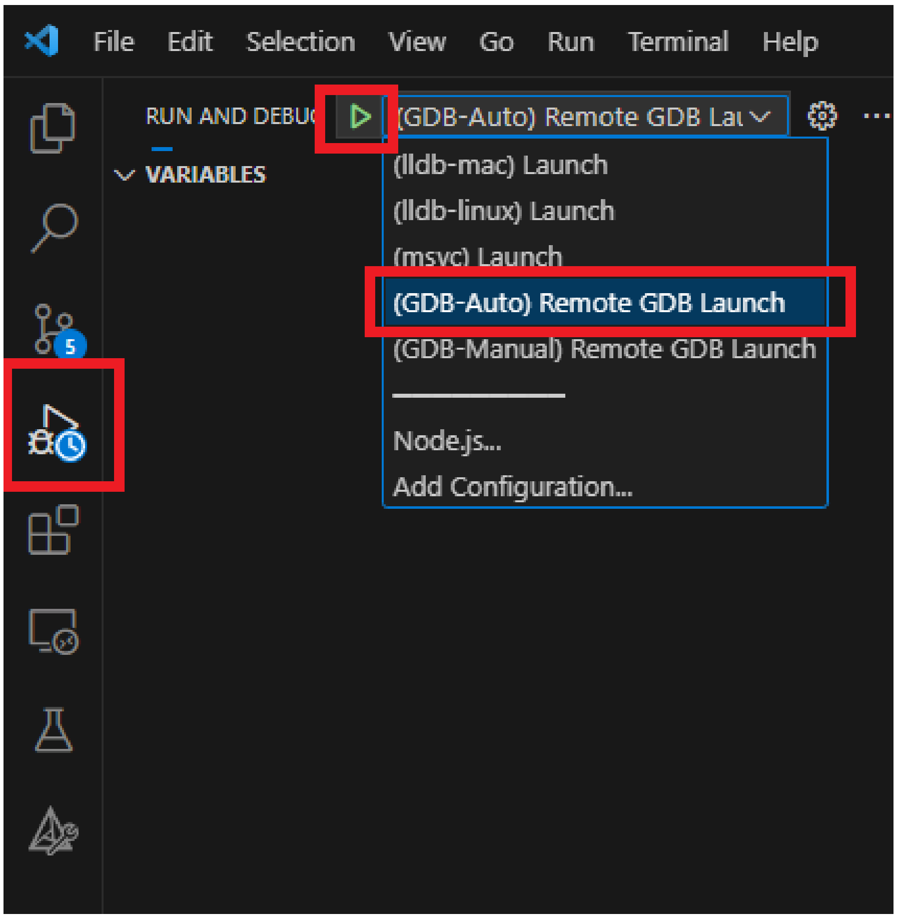
```
sudo ./launch.py ipaddress
```

This will generate the `launch.json` file in the `.vscode` folder. Now we need to install `gdbserver` on the **Target** machine. To do so, run the following command on **Host**:

```
ssh RPi0
sudo apt-get install gdbserver
```

### 6.9.2 Debugging

To debug the project, open the Debugger Tab (`Ctrl + Shift + D`) and select the profile `(GDB-Auto) Remote GDB Launch` and click on the Run button. This will start the `gdbserver` on the **Target** machine and connect to it. You should now be able to debug the project.



### 6.9.3 Known Issues

In some cases, certain libraries cannot be found. To fix this, run the following command on **Host**:

- `libncurses.so`

```
    sudo apt install libncurses5
```

- `libpython2.7.so`

```
    sudo apt install libpython2.7
```

## 6.10 Extra Features

In this section, we will explore some extra features that can be used to improve the development experience.

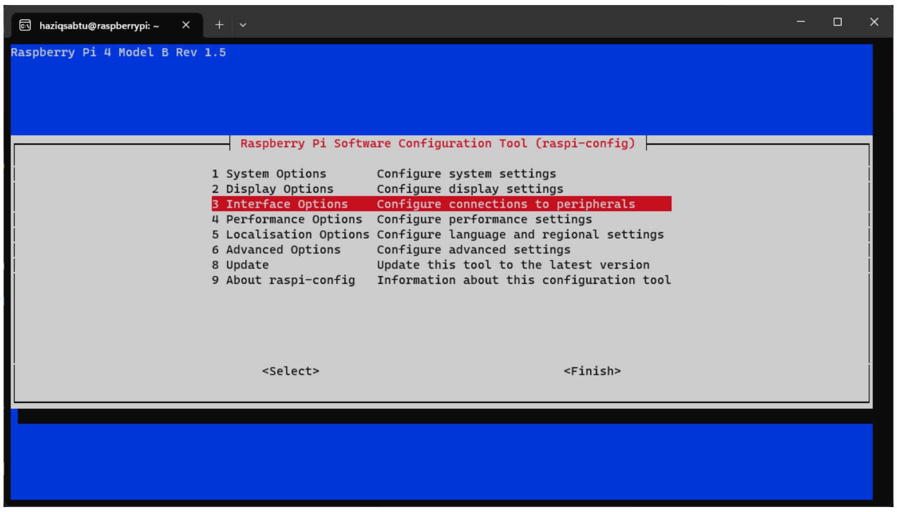
### 6.10.1 VNC Viewer

VNC Viewer is a remote desktop application that allows you to remotely control a computer. It is useful for accessing the **Target** machine remotely. By properly integrating the VNC, your Raspberry Pi can be used without a monitor, keyboard, or mouse.

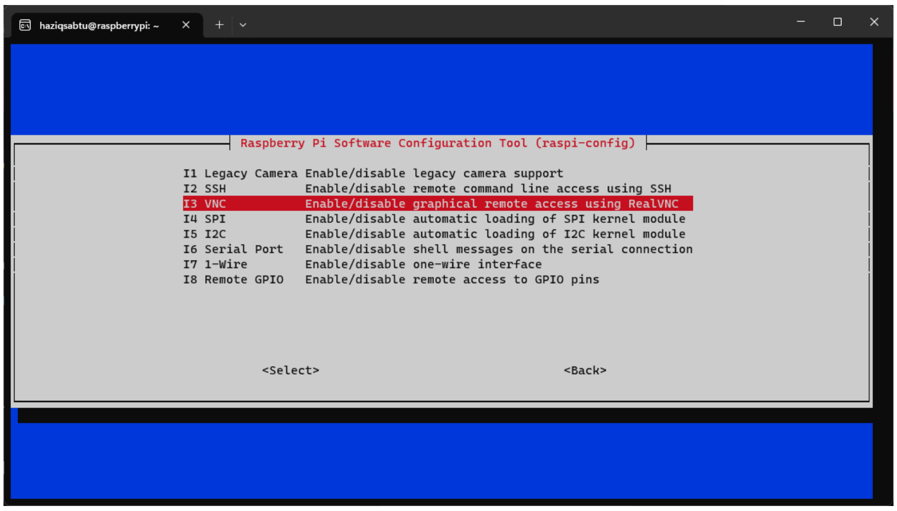
To download VNC Viewer, go to this link and download the installer for your operating system. Once installed, you may need to create an account to use the application. Once done you may need to prepare the **Target** machine to allow VNC connection. To do so, run the following command on **Host**:

```
ssh RPi0
sudo raspi-config
```

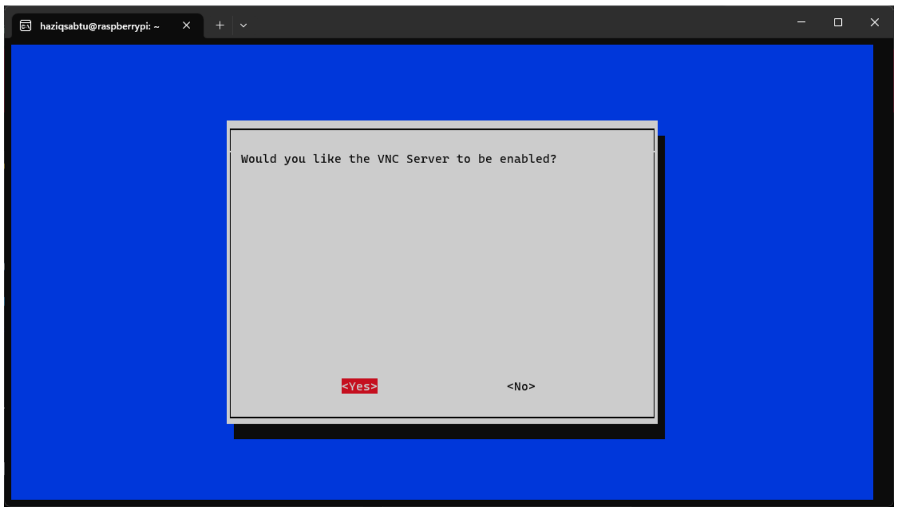
Navigate to **Interface Options** and enable **VNC**.



```
Raspberry Pi 4 Model B Rev 1.5

Raspberry Pi Software Configuration Tool (raspi-config)

1 System Options       Configure system settings
2 Display Options      Configure display settings
3 Interface Options    Configure connections to peripherals
4 Performance Options  Configure performance settings
5 Localisation Options Configure language and regional settings
6 Advanced Options     Configure advanced settings
8 Update               Update this tool to the latest version
9 About raspi-config   Information about this configuration tool

          <Select>                         <Finish>
```



```
Raspberry Pi Software Configuration Tool (raspi-config)

I1 Legacy Camera Enable/disable legacy camera support
I2 SSH           Enable/disable remote command line access using SSH
I3 VNC           Enable/disable graphical remote access using RealVNC
I4 SPI           Enable/disable automatic loading of SPI kernel module
I5 I2C           Enable/disable automatic loading of I2C kernel module
I6 Serial Port   Enable/disable shell messages on the serial connection
I7 1-Wire        Enable/disable one-wire interface
I8 Remote GPIO   Enable/disable remote access to GPIO pins

          <Select>                         <Back>
```



```
Would you like the VNC Server to be enabled?

<Yes>                    <No>
```

Now, you should be able to access the **Target** machine via VNC Viewer. It is recommended to reboot the **Target** machine after enabling VNC.

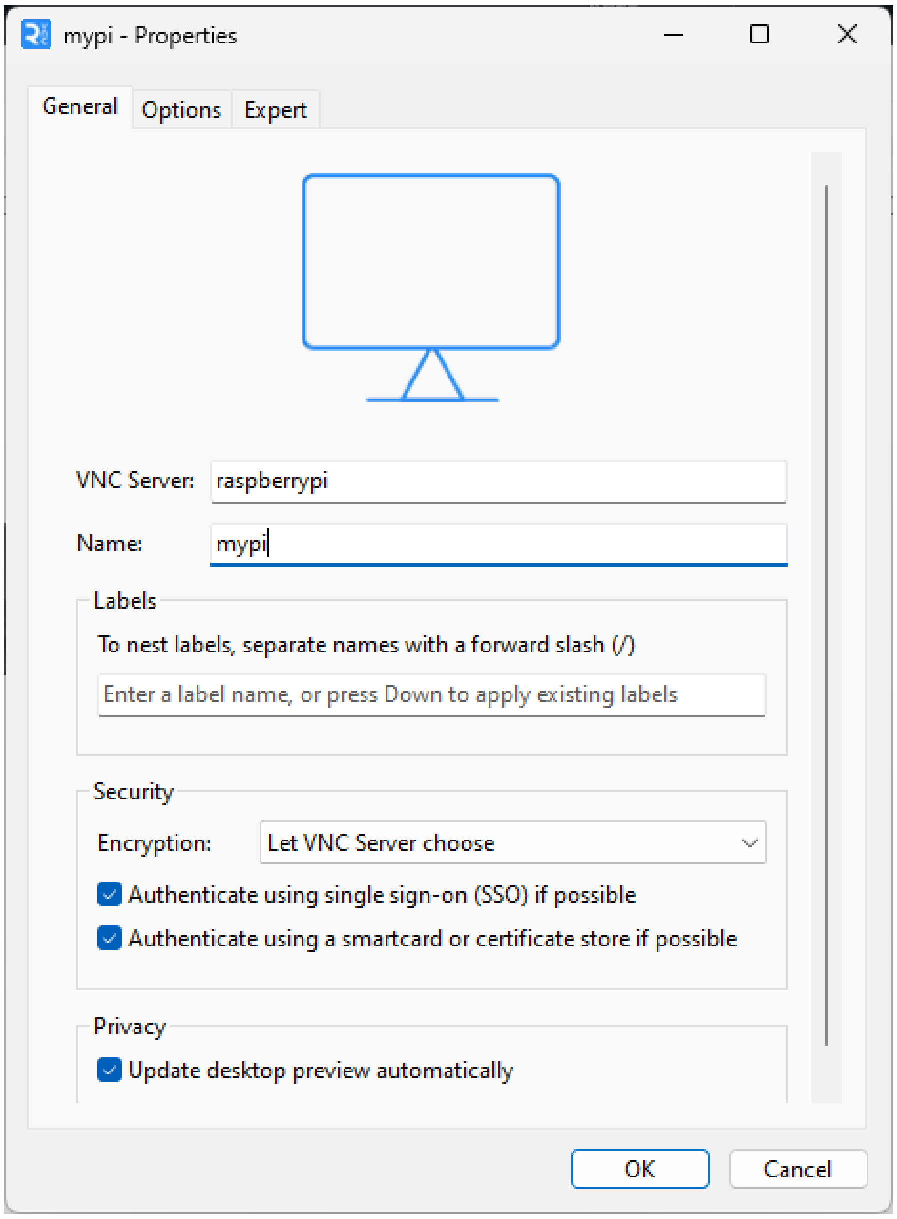
To access the **Target** machine via VNC Viewer, you need to know the IP address of the **Target** machine. To do so, run the following command on **Host**:

```
hostname -I
```

Alternatively, it is defaulted to `raspberrypi`.

Now, you can access the **Target** machine via VNC Viewer. To do so, open VNC Viewer and create a new connection (`Ctrl + N`) and set the following parameters:

- **VNC Server** - IP address of the **Target** machine
- **Name** - Name of the connection, can be anything


mypi - Properties

General | Options | Expert

VNC Server: raspberrypi

Name: mypi

Labels

To nest labels, separate names with a forward slash (/)

Enter a label name, or press Down to apply existing labels

Security

Encryption: Let VNC Server choose

✓ Authenticate using single sign-on (SSO) if possible

✓ Authenticate using a smartcard or certificate store if possible

Privacy

✓ Update desktop preview automatically

Now, you should be able to access the **Target** machine via VNC Viewer. You may need to enter the password of the **Target** machine.

### 6.10.2 VSCode Extensions

Following are some useful extensions that can be used to improve the development experience:

- Clang-Format: Clang-Format is a tool that formats C/C++/Obj-C code according to a set of style options, similar to the way that clang-format formats code.
- Github Copilot: GitHub Copilot is an AI pair programmer that helps you write code faster and with less work.
- Todo Tree: Todo Tree is a task manager that helps you manage your TODOs.

# 7 Native Compilation

For those who prefer to develop natively on the Raspberry Pi, this section will guide you through the process of setting up the development environment.

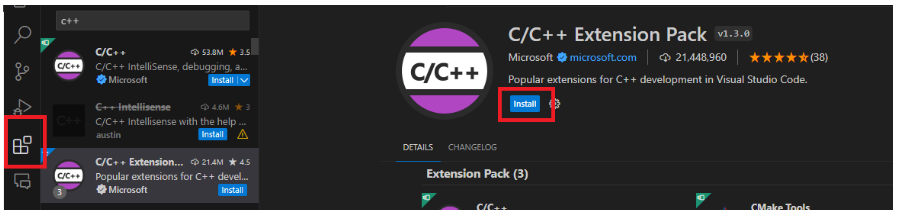
For native compilation, the steps are much simpler. Simply install the required dependencies on the **Target** machine and you are good to go. Refer to this section for more information.

The installation of VSCode on Raspberry Pi is done with this instruction.

## 7.1 Installing required Extensions

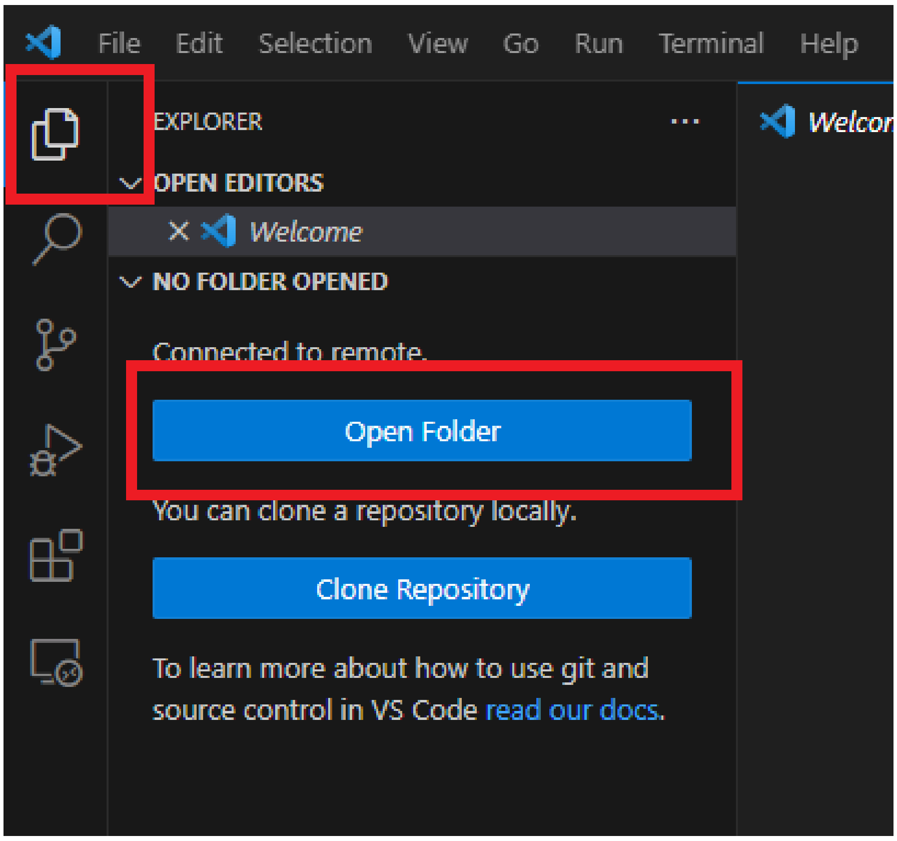
To install extensions, open Visual Studio Code and press `Ctrl+Shift+X` to open the Extensions pane. Search for the following extensions and install them:

- C/C++ Extension Pack


c++
C/C++
C/C++ IntelliSense, debugging, a...
Microsoft
Install
~~C++ Intellisense~~
C/C++ Intellisense with the help ...
austin
Install
C/C++ Extension...
Popular extensions for C++ devel...
Microsoft
Install
C/C++ Extension Pack v1.3.0
Microsoft microsoft.com
21,448,960
(38)
Popular extensions for C++ development in Visual Studio Code.
Install
DETAILS
CHANGELOG
Extension Pack (3)
C/C++
CMake Tools

## 7.2 Opening SpeedCameraPi project

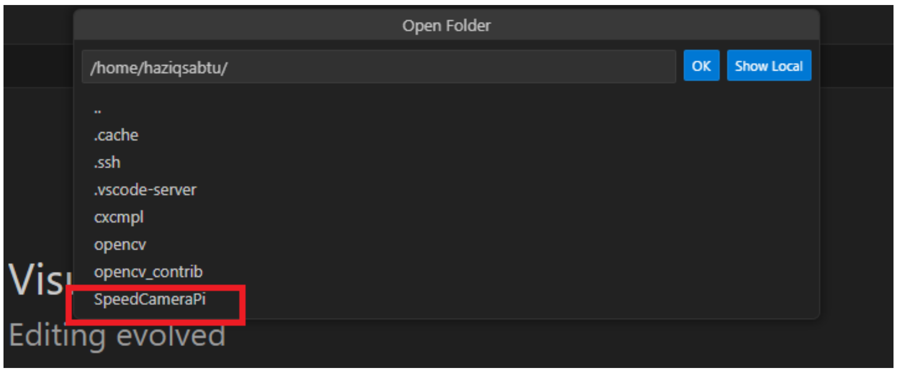
To open the SpeedCameraPi project, open the Explorer pane (`Ctrl+Shift+E`) and click on the Open Folder button. Navigate to the SpeedCameraPi folder and click on the OK button. You should now see the SpeedCameraPi project in the Explorer pane.



Connected to remote.

You can clone a repository locally.

To learn more about how to use git and source control in VS Code read our docs.

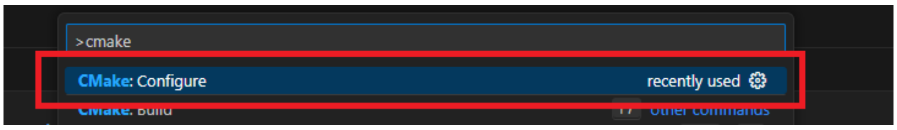


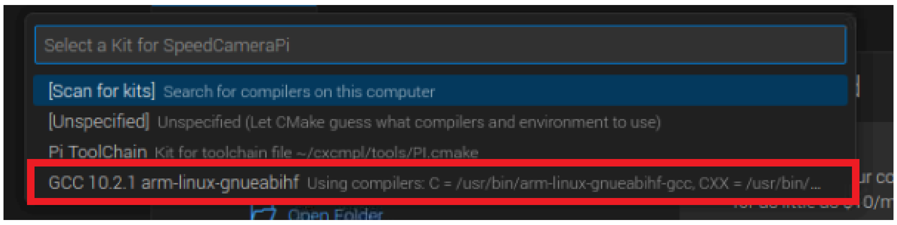
/home/haziqsabtu/

..
.cache
.ssh
.vscode-server
cxcmpl
opencv
opencv_contrib
SpeedCameraPi

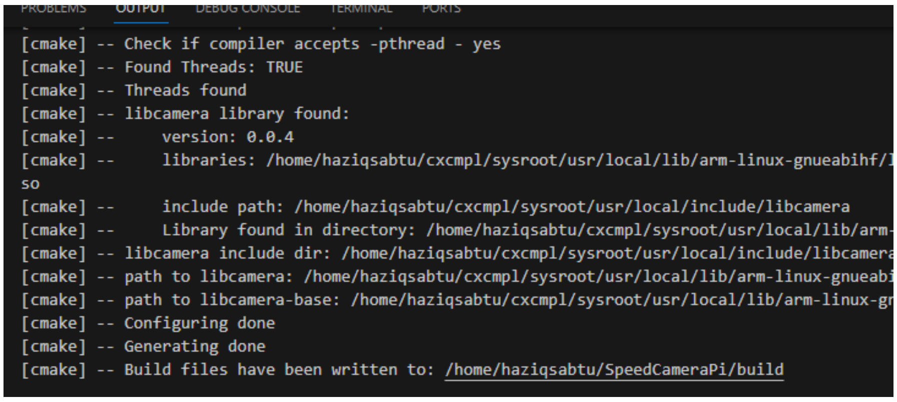
## 7.3 Configuring C/C++ extension

To configure the C/C++ extension, open the Command Palette (`Ctrl+Shift+P`) and type `CMake:Configure` and select the default compiler. This will create a `build` folder in the project directory and generate the CMake cache.




Select a Kit for SpeedCameraPi
[Scan for kits] Search for compilers on this computer
[Unspecified] Unspecified (Let CMake guess what compilers and environment to use)
Pi ToolChain Kit for toolchain file ~/cxcmpl/tools/PI.cmake
GCC 10.2.1 arm-linux-gnueabihf Using compilers: C = /usr/bin/arm-linux-gnueabihf-gcc, CXX = /usr/bin/...
Open Folder

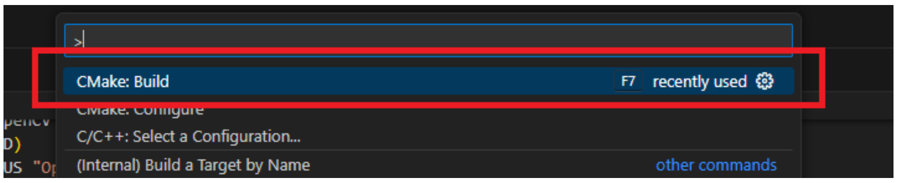
If the configuration process is successful, you should see the following message in the terminal:



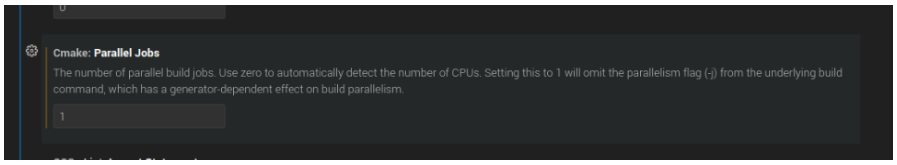
```
[cmake] -- Check if compiler accepts -pthread - yes
[cmake] -- Found Threads: TRUE
[cmake] -- Threads found
[cmake] -- libcamera library found:
[cmake] --     version: 0.0.4
[cmake] --     libraries: /home/haziqsabtu/cxcmpl/sysroot/usr/local/lib/arm-linux-gnueabihf/l
so
[cmake] --     include path: /home/haziqsabtu/cxcmpl/sysroot/usr/local/include/libcamera
[cmake] --     Library found in directory: /home/haziqsabtu/cxcmpl/sysroot/usr/local/lib/arm-
[cmake] -- libcamera include dir: /home/haziqsabtu/cxcmpl/sysroot/usr/local/include/libcamera
[cmake] -- path to libcamera: /home/haziqsabtu/cxcmpl/sysroot/usr/local/lib/arm-linux-gnueabi
[cmake] -- path to libcamera-base: /home/haziqsabtu/cxcmpl/sysroot/usr/local/lib/arm-linux-gr
[cmake] -- Configuring done
[cmake] -- Generating done
[cmake] -- Build files have been written to: /home/haziqsabtu/SpeedCameraPi/build
```

## 7.4 Building the project

To build the project, open the Command Palette (`Ctrl+Shift+P`) and type `CMake:Build`. This will build the project and generate the executable file in the `build` folder.


>|
CMake: Build
F7
recently used
CMake: Configure
C/C++: Select a Configuration...
(Internal) Build a Target by Name
other commands

If the UI is unresponsive, it is highly likely that the building process is using all of the resources available. Either wait until the process completes, or reduce the number of parallel build jobs. To do so, open the Command Palette (`Ctrl+Shift+P`) and type `Settings: Open Settings (UI)`. Navigate to `CMake: Parallel Jobs` and set the value to 1.

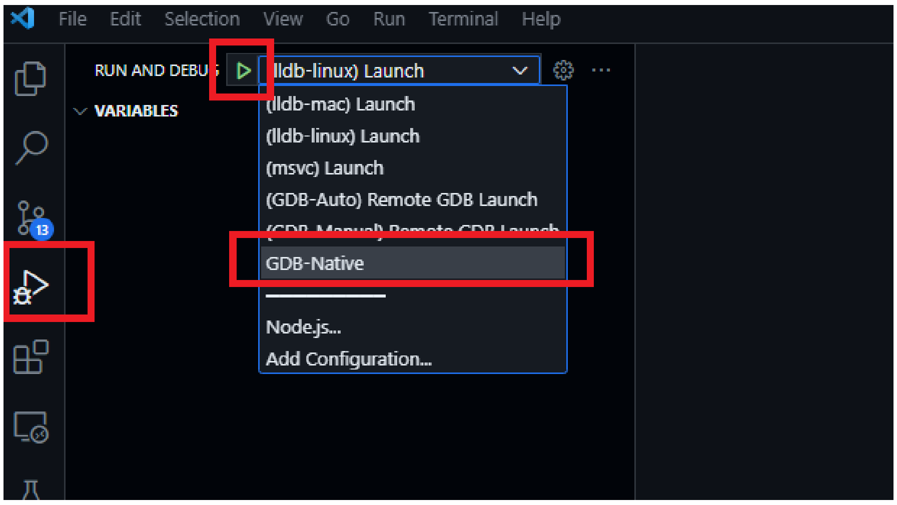

Cmake: **Parallel Jobs**

The number of parallel build jobs. Use zero to automatically detect the number of CPUs. Setting this to 1 will omit the parallelism flag (-j) from the underlying build command, which has a generator-dependent effect on build parallelism.

1

## 7.5 Debugging

To debug the project, open the Debugger Tab (`Ctrl + Shift + D`) and select the profile `GDB - Native` and click on the Run button. This will compile the project and start the debugger. You should now be able to debug the project.



# Part III

# Getting Started with Development

# 8 Introduction

This section will guide you through the process of developing the application. It is assumed that you have already set up the development environment. If not, please refer to the previous section. This section will cover the following topics:

- **Adding a new Panel** - This section will guide you through the process of adding a new panel to the application.
- **Custom Event** - This section will guide you through the process of defining, triggering, and handling custom events.
- **Defining Process Threads** - This section will guide you through the process of defining process threads.
- **Creating Tasks** - This section will guide you through the process of creating tasks for ThreadPool.

# 9 Add new Panel

This chapter will provide description on how to add new panel to the application. To add a new Panel to the Application, following steps are required:

1. Define a new PanelID Enum
2. Create a new Controller class
3. Create a new Panel class
4. Define object in MainFrame

## 9.1 Define a new PanelID Enum

The first step is to register a unique enum for the Panel. This enum will be used to identify the Panel in the application. To add a new Panel enum, modify the PanelID enum in the `include/Model/Enum.hpp` file. Following example will provide a simple example of a PanelID enum:

**Listing 9.1:** PanelID enum example

```
enum PanelID {
    // ... other enum
    PANEL_INFO,
    PANEL_MY_PANEL, // Add new PanelID enum here
};
```

## 9.2 Create a new Controller class

### 9.2.1 Define a new Controller class

To create a new controller class, you first need to decide either the Panel will require a touch input or not. If touch input is required you can create a custom controller class that extends the BaseControllerWithTouch class. Otherwise you can create a custom controller class that extends the BaseController class. Following example will provide a simple example of a Controller class that extends the BaseControllerWithTouch class:

**Listing 9.2:** MyController class example

```cpp
class MyController : public BaseControllerWithTouch {
  public:
    MyController(ResourcePtr shared);
    ~MyController();

  private:
    static const PanelID currentPanelID = PanelID::PANEL_MY_PANEL; // Add new PanelID enum here

  private:
    void throwIfAnyThreadIsRunning() override;

    void killAllThreads(wxEvtHandler *parent) override;

    void leftDownHandler(wxEvtHandler *parent, cv::Point point) override;
    void leftMoveHandler(wxEvtHandler *parent, cv::Point point) override;
    void leftUpHandler(wxEvtHandler *parent, cv::Point point) override;
};
```

Make sure that you implemented the pure virtual functions from the BaseControllerWithTouch class. The pure virtual functions are:

- `void throwIfAnyThreadIsRunning() override;`
- `void killAllThreads(wxEvtHandler *parent) override;`
- `void leftDownHandler(wxEvtHandler *parent, cv::Point point) override;`
- `void leftMoveHandler(wxEvtHandler *parent, cv::Point point) override;`
- `void leftUpHandler(wxEvtHandler *parent, cv::Point point) override;`

See the source code for `LaneCalibrationController` for example.

To make the code more readable, it is advised to define the controller shared pointer to a shorter name. Following example will provide a simple example of defining a shorter name for the Controller shared pointer:

**Listing 9.3:** Shorter name for Controller shared pointer

```
#define MCPtr std::shared_ptr<MyController>
```

### 9.2.2 Add method to ControllerFactory

The `ControllerFactory` is used to connect the Controller with the Model component. Following example can be used:

**Listing 9.4:** ControllerFactory method example

```
class ControllerFactory {
  public:
    ControllerFactory(wxWindow *parent);
    ~ControllerFactory();

    ResourcePtr getSharedModel();

    // ... other methods
    MyPtr createMyController();

  private:
    ResourcePtr shared;
};
```

**Listing 9.5:** Implementation of createMyController method

```
MyPtr ControllerFactory::createMyController() {
    return std::make_shared<MyController>(shared);
}
```

## 9.3 Create a new Panel class

### 9.3.1 Define a new Panel class

To create a new Panel class, you need to create a new class that extends either the BasePanel or the BasePanelWithTouch class. The BasePanel class has already implemented the basic functionality of a Panel, while the BasePanelWithTouch class has already implemented the basic functionality of a Panel with touch support. The BasePanelWithTouch class is recommended to be used if the Panel will be used in a touch screen device. Alternatively you can also create a whole new class that extends the wxPanel class from wxWidgets. Following example will provide a simple example of a Panel class that extends the BasePanelWithTouch class:

**Listing 9.6:** MyPanel class example

```
class MyPanel : public BasePanelWithTouch {
  public:
    MyPanel(wxWindow *parent, wxWindowID id, MyPtr controller);
    ~MyPanel();

  private:
    const PanelID panel_id = PANEL_MY_PANEL;

    MyPtr controller;

    DECLARE_EVENT_TABLE()
};
```

Make sure that if you are using the BasePanelWithTouch class, you are also using the BaseControllerWithTouch class. Otherwise you will get a compilation error.

### 9.3.2 Create ButtonPanel

Now create a ButtonPanel for the Panel. The ButtonPanel is a panel that contains buttons to perform certain actions. To create a ButtonPanel, you need to create a new class that extends the BaseButtonPanel class. Following example will provide a simple example of a ButtonPanel class that extends the BaseButtonPanel class:

**Listing 9.7:** CustomButtonPanel class example

```
class CustomButtonPanel : public BaseButtonPanel {
  public:
    CustomButtonPanel(wxWindow *parent, wxWindowID id);

    void update(const AppState &state) override;

    // define all buttons here

    DECLARE_EVENT_TABLE();
};
```

Make sure that the `update()` method is implemented. The `update()` method will be called by the Panel to update the state of the buttons.

### 9.3.3 Implement the Panel class

Now that you have created the ButtonPanel, you can now implement the Panel class. Following example can be used as a reference to implement the Panel class:

**Listing 9.8:** Implementation of MyPanel class

```
MyPanel::MyPanel(wxWindow *parent, wxWindowID id,
                                            MyPtr controller)
    : BasePanelWithTouch(parent, id, controller), controller(
        controller) {

    button_panel =
        new CustomButtonPanel(this, wxID_ANY);
    title_panel = new TitlePanel(this, panel_id);

    size();
}
```

Make sure the `size()` method is called at the end of the constructor. The `size()` method will set the sizer of the Panel.

### 9.3.4 Add method to PanelFactory

In this step, you need to modify the PanelFactory class. Following example can be used:

**Listing 9.9:** Create Panel method in PanelFactory

```
wxPanel *createPanel(wxWindow *parent, PanelID panelID) {
        // ... other if case

        if (panelID == PANEL_MY_PANEL) {
            return createMyPanel(parent);
        }
    }

MyPanel *createMyPanel(wxWindow *parent) {
        return new MyPanel(parent, wxID_ANY, controllerFactory->
            createMyController());
    }
```

## 9.4 Define object in MainFrame

The last step is to define the Panel in the MainFrame class. Following example can be used:

**Listing 9.10:** MainFrame class example

```
class MainFrame : public wxFrame {
    // ... other methods

    private:
    // ... other methods
    MyPanel *myPanel;
    // ... other methods
};

MainFrame::MainFrame() : wxFrame(NULL, wxID_ANY, Data::AppName) {

    // ... other methods

    // ... other registerPanel
    registerPanel(PANEL_MY_PANEL);

    showFirstPanel();
}
```

Now you have successfully added a new Panel to the application.

# 10 Custom Event

Events are used as a response mechanism for the View component. There are two types of events in this application:

- DataEvent
- EmptyEvent

## 10.1 DataEvent

DataEvent is an event that contains data. The data can be any type of data. To create a new DataEvent, you need to create a new class that extends the `wxCommandEvent` class. Following example will provide a simple example of a DataEvent class:

**Listing 10.1:** CustomEvent class example

```
class CustomEvent;
wxDECLARE_EVENT(c_CUSTOM_EVENT, CustomEvent);

enum CUSTOM_EVENT_TYPE {
    CUSTOM_EVENT_TYPE_1 = 1,
    CUSTOM_EVENT_TYPE_2,
    CUSTOM_EVENT_TYPE_3,
};

class CustomEvent : public wxCommandEvent {
  public:
    CustomEvent(wxEventType type, int id = 1);

    CustomEvent(const CustomEvent &event);
    virtual wxEvent *Clone() const override;

    // Define set and get method for the data here
    void setData(const std::string &data);
    std::string getData() const;

  private:
    std::string data;
};

// this part is important
typedef void (wxEvtHandler::*CustomFunction)(CustomEvent &);
#define CustomHandler(func)                                          \
    wxEVENT_HANDLER_CAST(CustomFunction, func)
#define EVT_CUSTOM(id, func)                                       \
    wx__DECLARE_EVT1(c_CUSTOM_EVENT, id, CustomHandler(func))
```

**Listing 10.2:** CustomEvent continued

```
wxDEFINE_EVENT(c_CUSTOM_EVENT, CustomEvent);

CustomEvent::CustomEvent(wxEventType type, int id)
    : wxCommandEvent(type, id) {
}

CustomEvent::CustomEvent(const CustomEvent &event)
    : wxCommandEvent(event) {
    this->setData(event.getData());
}

wxEvent *UpdatePreviewEvent::Clone() const {
    return new CustomEvent(*this);
}

void CustomEvent::setData(const std::string &data) {
    this->data = data;
}

std::string CustomEvent::getData() const {
    return data;
}
```

### 10.1.1 Example Classes

The following classes can be referred to as examples:

- UpdatePreviewEvent
- UpdateStateEvent
- ErrorEvent

## 10.2 EmptyEvent

EmptyEvent is an event that does not contain any data. It is normally used to signal the View component to perform certain action. To create a new EmptyEvent, you need to create a new class that extends the `wxCommandEvent` class. Following example will provide a simple example of an EmptyEvent class:

**Listing 10.3:** EmptyEvent class example

```
wxDECLARE_EVENT(c_CUSTOM_EVENT, wxCommandEvent);

enum CUSTOM_EVENT_TYPE {
    CUSTOM_EVENT_TYPE_1 = 1,
    CUSTOM_EVENT_TYPE_2,
    CUSTOM_EVENT_TYPE_3,
};
```

**Listing 10.4:** EmptyEvent continued

```
wxDEFINE_EVENT(c_CUSTOM_EVENT, wxCommandEvent);
```

## 10.3 Bind Event

To bind an event to a method, you need to use the following syntax in the Panel class:

**Listing 10.5:** Binding Event in Panel class

```
BEGIN_EVENT_TABLE(BasePanel, wxPanel)
    // ... other event

    // For EmptyEvent
    EVT_COMMAND(wxID_ANY, c_CUSTOM_EVENT, BasePanel::OnCustomEvent)

    // For DataEvent
    EVT_CUSTOM(c_CUSTOM_EVENT, BasePanel::OnCustomEvent)
END_EVENT_TABLE()
```

Make sure the function to handle the event is implemented.

## 10.4 Submit Event

To submit an event, you need to use the following syntax:

**Listing 10.6:** Submitting Event

```
// For EmptyEvent
wxCommandEvent event(c_CUSTOM_EVENT, CUSTOM_EVENT_TYPE_1);
wxPostEvent(this, event);

// For DataEvent
CustomEvent event(c_CUSTOM_EVENT, CUSTOM_EVENT_TYPE_1);
event.setData("data");
wxPostEvent(this, event);
```

# 11 Thread

Thread is used to perform a long-running task.

## 11.1 Define unique ThreadID

To define a unique ThreadID, you need to modify the ThreadID enum in the `include/Thread/Thread_ID.hpp` file. Following example will provide a simple example of a ThreadID enum:

**Listing 11.1:** ThreadID enum example

```
enum ThreadID {
    // ... other enum
    THREAD_MY_THREAD, // Add new ThreadID enum here
};
```

## 11.2 Create a new Thread class

Now you need to create a new class that extends the `BaseThread` class. Following example will provide a simple example of a Thread class:

**Listing 11.2:** CustomThread class example

```
class CustomThread : public BaseThread {
  public:
    CustomThread(wxEvtHandler *parent, DataPtr data);
    ~CustomThread();

    ThreadID getID() const override;

  protected:
    virtual ExitCode Entry() override;

  private:
    // ... other methods
    const ThreadID thread_id = ThreadID::THREAD_MY_THREAD; // Add new
         ThreadID enum here
};

CustomThread::CustomThread(wxEvtHandler *parent, DataPtr data)
    : BaseThread(parent, data) {
}

CustomThread::~CustomThread() {
}

ThreadID CustomThread::getID() const {
    return thread_id;
}

wxThread::ExitCode CustomThread::Entry() {
    // ... other code, do process
    return (wxThread::ExitCode)0;
}
```

Additionally, following classes can be inherited to add more functionality:

| Class | Description |
|---|---|
| PreviewableThread | Enable the thread to send image to ImagePanel |
| ImageSizeDataThread | Add variable `imageSize`, which is the size of the captured image |
| ImageDataThread | Add variable `image`, which is the camera |
| CameraAccessor | Enable camera access |

## 11.3 Add method to ThreadController

In this step, you need to modify the `ThreadController` class. Following example can be used:

**Listing 11.3:** ThreadController class example

```
class ThreadController {
  public:
    ThreadController(wxEvtHandler *parent, DataPtr data);
    ~ThreadController();

    // ... other methods

    virtual void startCustomThread(wxEvtHandler *parent, PanelID
       panelID);

    void endCustomThread();

    CustomThread *getCustomThread();

  private:
    // ... other variables
    CustomThread *custom_thread;
};
```

# 12 Task for ThreadPool

ThreadPool enables parallel processing, which can be used to speed up the application. To add a new Task to the ThreadPool, following steps are required:

## 12.1 Define unique TaskType

To define a unique TaskType, you need to modify the TaskType enum in the `include/Thread/Task/Task.hpp` file. Following example will provide a simple example of a TaskType enum:

**Listing 12.1:** TaskType enum example

```
enum TaskType {
    // ... other enum
    TASK_MY_TASK, // Add new TaskType enum here
};
```

## 12.2 Create a new Task class

Now you need to create a new class that extends the Task class. Following example will provide a simple example of a Task class:

**Listing 12.2:** CustomTask class example

```
class CustomTask : public Task {
  public:
    CustomTask(wxEvtHandler *parent, DataPtr data);
    void Execute() override;

  private:
    // ... other methods
    const std::string currentName = "CustomTask";
    const TaskType currentType = TaskType::TASK_MY_TASK; // Add new
        TaskType enum here
};

CustomTask::CustomTask(wxEvtHandler *parent, DataPtr data)
    : Task(parent, data) {
    property = TaskProperty(currentType);
    name = currentName;
}

void CustomTask::Execute() {
    // ... other code, do process
}
```