SG Series

H2 API
EXAMPLE USAGE GUIDE

V1.1
05/21/2026

AROGIC

Contents

1. Version Management

Updated Description Sheet

Version	Description	Date
V1.1	1. Added: MATLAB example usage instructions for the Windows 2. Added: Example usage instructions for the Linux	05/21/2026
V1.0	1. Initial Version	04/30/2025

2. Windows System

2.1 C/C++

2.1.1 Configure Development Environment

The following uses configuring Debug Win32 as an example. For other build types and platforms, please refer to the end of this chapter.

1. Open Visual Studio 2019 and create a new project (eg, test);
2. After creation is complete, copy the "Windows\api" folder from the delivery USB drive to a directory at the same level as the project;

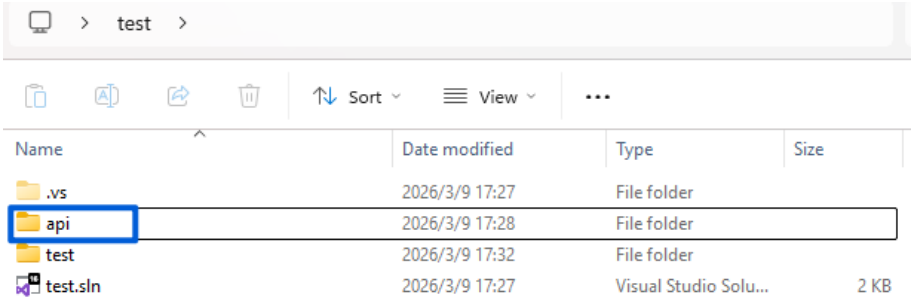

Figure 1 Copy the api file

3. Launch Visual Studio 2019, right-click on "Source Files" -> "Add" -> "New Item" -> "C++ File (.cpp)" to create a new "main.cpp" file;
4. Click "Project" -> "Properties" in the menu bar, set "Configuration" to "Debug" and "Platform" to "Win32";
5. In Configuration Properties -> Debugging, set the Environment variable to "Path=..\api\x86";

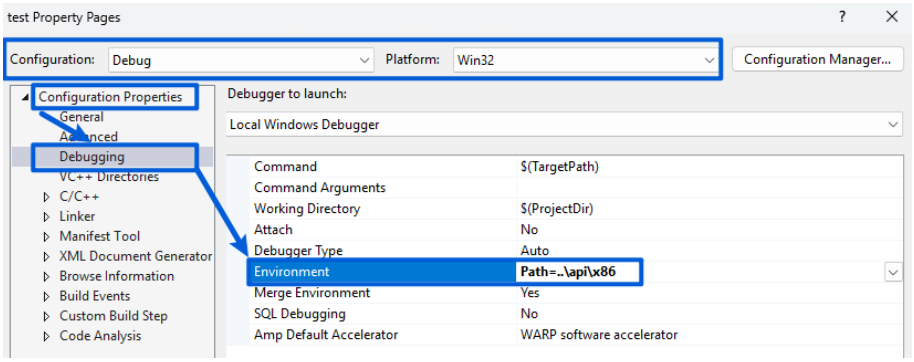

Figure 2 Configure environment variables

6. In Configuration Properties -> C/C++ -> General, set the Additional Include Directories to "$(SolutionDir)\api\x86";

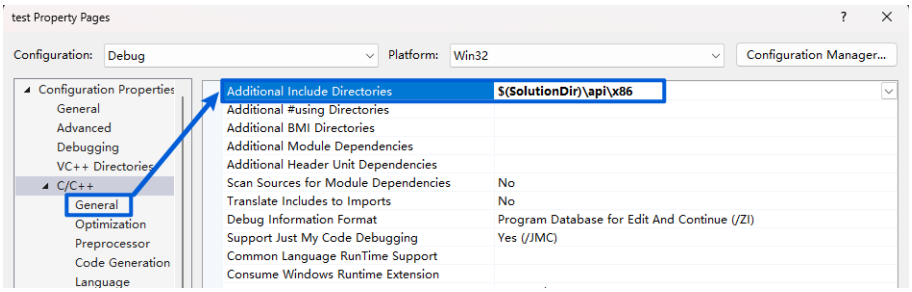

Figure 3 Add additional include directories

7. Set the "Additional Library Directories" under Configuration Properties -> Linker -> General to "$(SolutionDir)\api\x86";

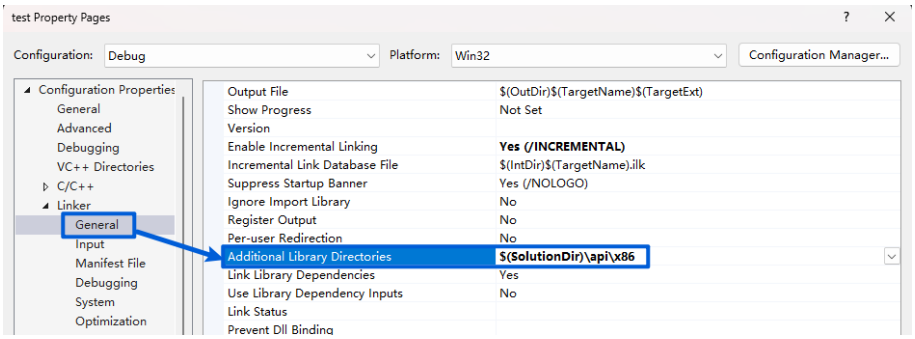

Figure 4 Configure additional library directories

8. Add "h2_api.lib" to "Additional Dependencies" under Configuration Properties -> Linker -> Input, then click "OK".

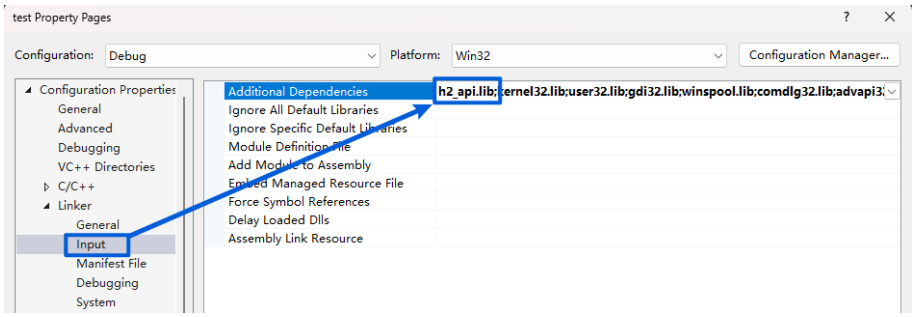

Figure 5 Configure additional dependencies

- Configure Release Win32: Set "Configuration" in Step 4 to "Release", and keep all other settings unchanged;
- Configure Debug x64: Set "Configuration" in Step 4 to "Debug" and "Platform" to "x64". Also replace all environment variables or additional directory paths containing "api\x86" with "api\x64";
- Configure Release x64: Set "Configuration" in Step 4 to "Release" and "Platform" to "x64". Also replace all environment variables or additional directory paths containing "api\x86" with "api\x64";

2.1.2 Usage Process for C++ Examples

Note: Only one example can run at a time, and the example cannot run simultaneously with SAStudio4.

1. Use Visual Studio 2019 or later to open the solution "C++_Examples.sln" located in the folder "Windows\example\C++" on the provided USB drive;
2. In the Solution Explorer on the right, double-click "main.cpp" under the source file directory of the "C++_Example.sln" project.
3. Each example function is encapsulated in a independent function. To use one, uncomment the corresponding routine, save the file, select the build architecture, and then click Run.

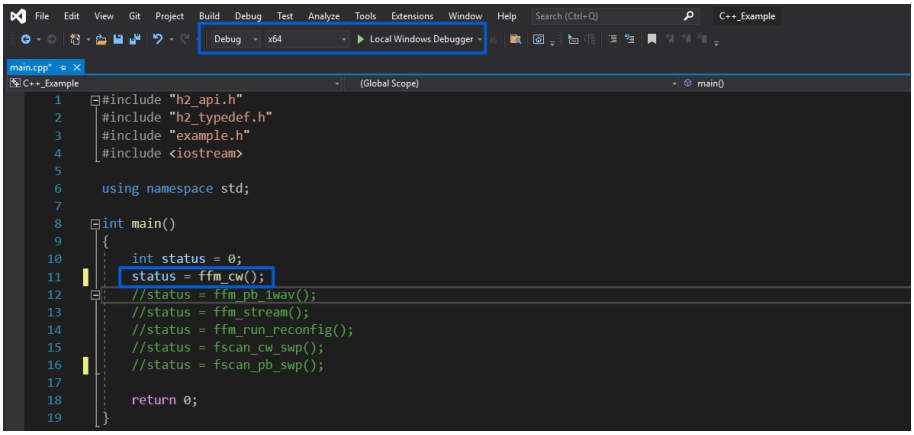

Figure 6 Run the ffm_cw example

2.2 QT

2.2.1 Configure the development environment

The following uses configuring an x64 architecture development environment as an example:

1. Create a new folder (referred to as *QtTest* below) to store project files (make sure the path does not contain Chinese characters);
2. Copy the contents of the “Windows\api\x64” folder from the provided USB drive into the newly created “QtTest\api” folder;
3. Open Qt Creator and click “File” -> “New File or Project”;
4. Under Projects, select “Application (Qt)”, choose “Qt Widgets Application”, and click “Choose...”;
5. Enter the project name (e.g., *test*), click “Browse”, select the previously created *QtTest* folder, then click “Next”;
6. Select “qmake”, continue clicking “Next” until reaching the “Kit Selection” page. On this page, choose a build kit, then click “Next”;

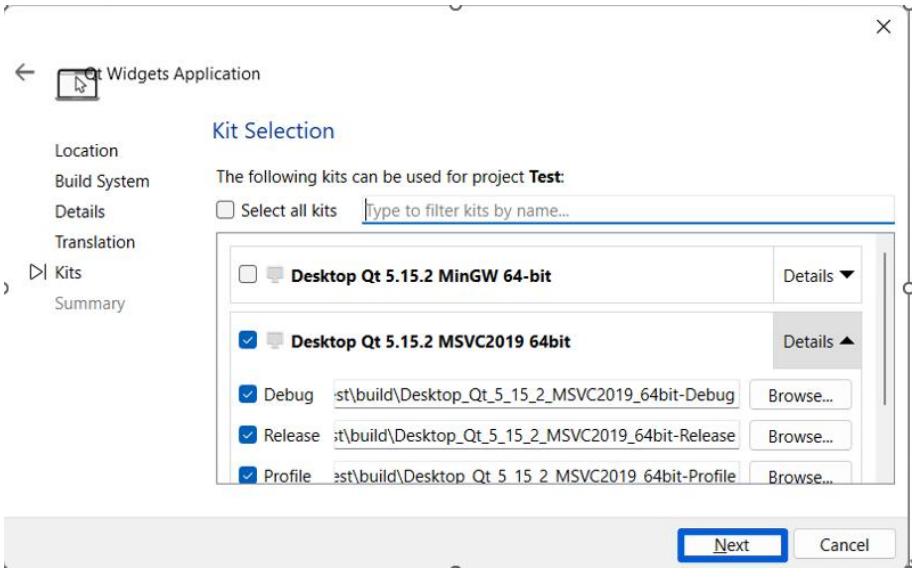

Figure 7 Selecting the Build Environment

7. Click “Finish” to create the project;
8. In the Qt Creator main interface, right-click the “test” project, then select “Add Library...” → “External Library” → “Next”;
9. Click “Browse” for the library file, select the *h2_api.lib* file in “QtTest\api”, and click “Open”;
10. Uncheck all options under Windows, then select “Static Library”, choose the Windows platform, and click “Next”.

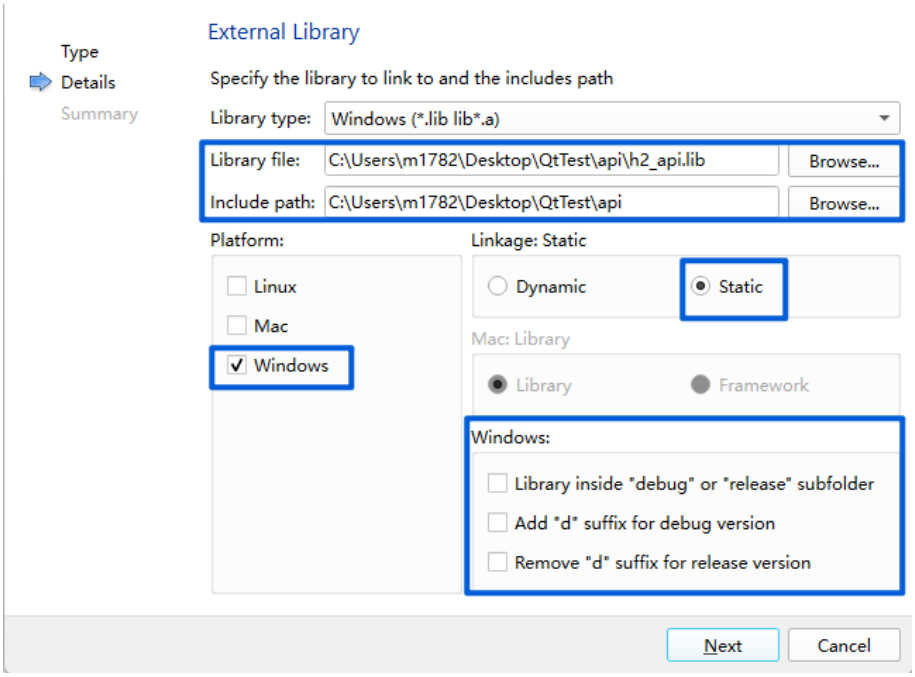

Figure 8 Adding an External Library

11. Click “Finish” to add the external library. Then delete the last line in the “test.pro” file: “else:win32-g++: PRE_TARGETDEPS += $$PWD/../api/libh2_api.a”. After saving, you can proceed with normal project development”;

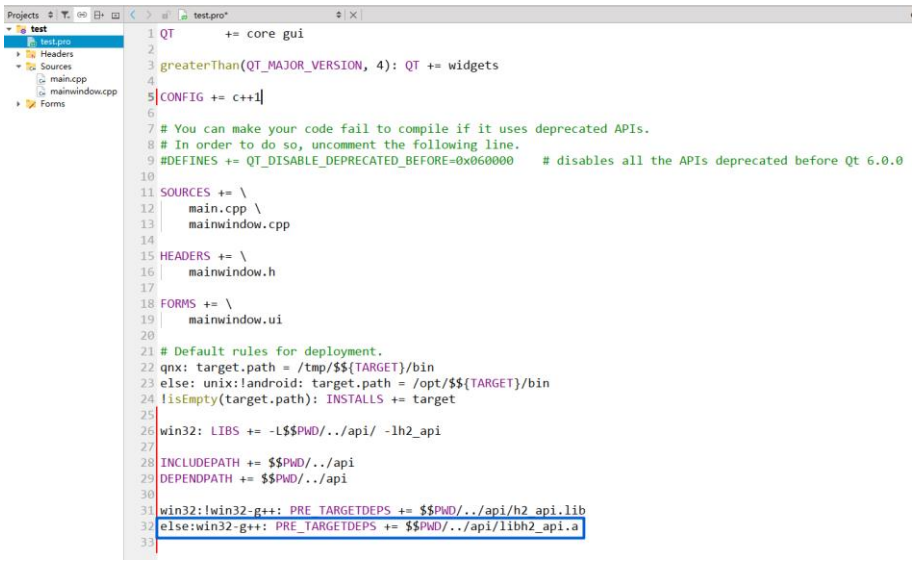

Figure 9 Modifying the test.pro File

12. After completing the code, click Run. The program running correctly is shown below.

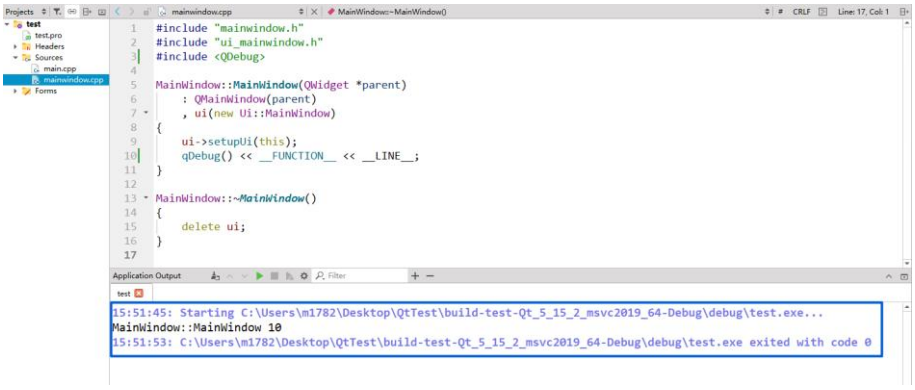

Figure 10 Correct Running Illustration

2.2.2 Example Usage Procedure

Note: Only one example can be run at a time. The example and the software cannot run simultaneously.

All Qt examples in the provided USB drive follow the same workflow. The following uses the *Qt_Example* project as an example:

1. Use Qt Creator to open the *Qt_Example.pro* file located in "Windows\examples\Qt\Qt_Example" on the provided USB drive (ensure the project path does not contain Chinese characters);
2. Click "Projects" and configure a 64-bit build kit for the project (the provided examples use 64-bit library files. If a 32-bit build environment is required, replace the libraries in the "Qt\api" folder with the 32-bit libraries from "\Windows\api\x86");

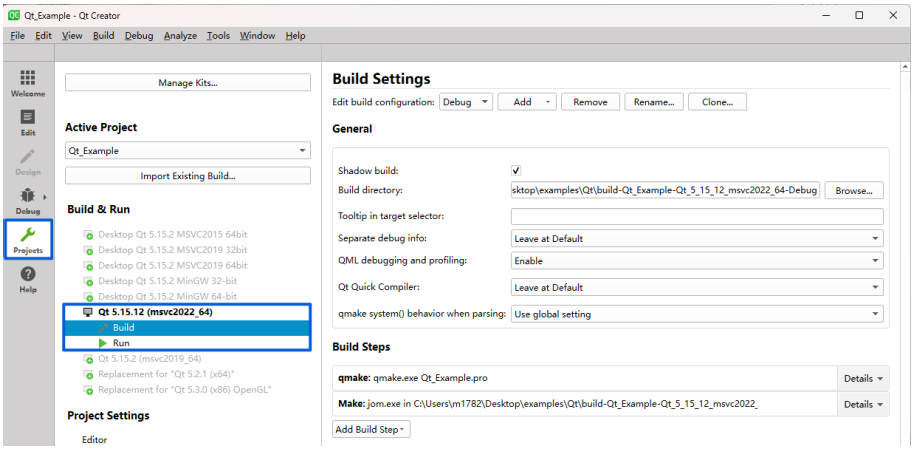

Figure 11 Configuring a 64-bit Build Environment

3. After the environment is configured, click “Edit” and open *main.cpp* under the “Sources” folder in the *Qt_Example* project. Uncomment the relevant functions and save the file. Then click Run to execute different examples.

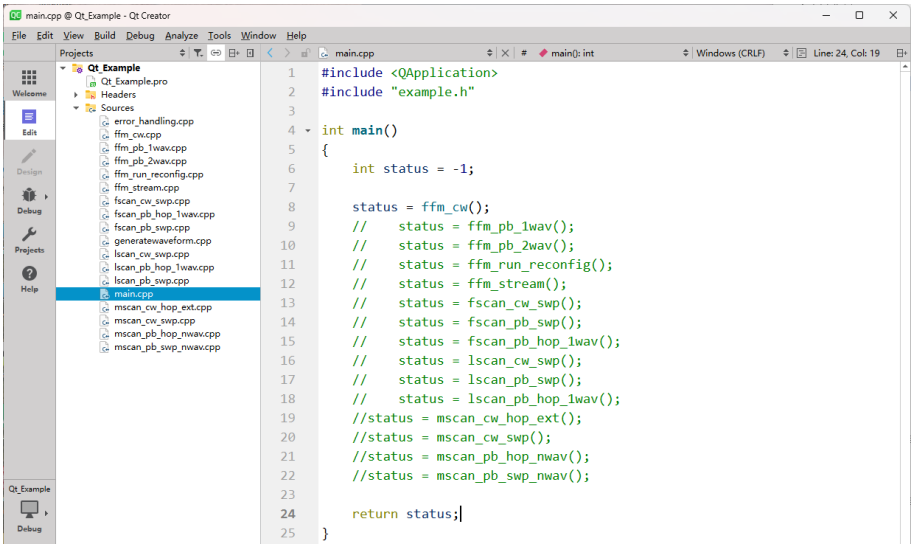

Figure 12 Run Qt Example

2.3 Python

2.3.1 Configuring the Development Environment

1. Create a new folder on the desktop (e.g., *test*);
2. Open the provided USB drive, navigate to “\Windows\examples\Python”, and copy the *api* folder and *h2_api.py* file into the newly created *test* folder;
3. Open Visual Studio Code, click “File” → “Open Folder”, and select the *test* folder;
4. In the Explorer panel on the left, click the “New File” icon to create a new Python file (e.g., *test.py*). You can then proceed to write code normally.

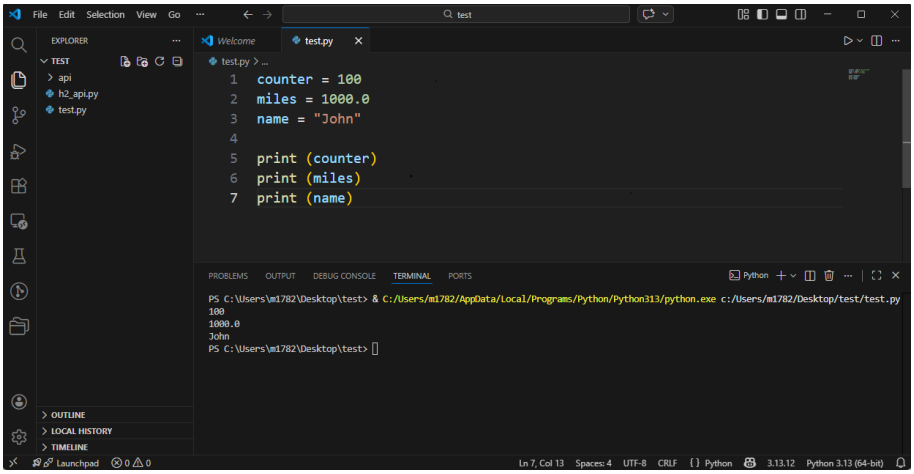

Figure 13 Creating a New Python Project

2.3.2 Example Usage Procedure

1. Use Visual Studio Code or another IDE to open the “Windows\examples\Python” project folder from the provided USB drive. The *h2_api.py* file is the Python mapping for the dynamic library, and the other files are example scripts;
2. After properly configuring the Python environment, select any example script (e.g., *ffm_cw.py*) and run it directly. The program running successfully is shown below;

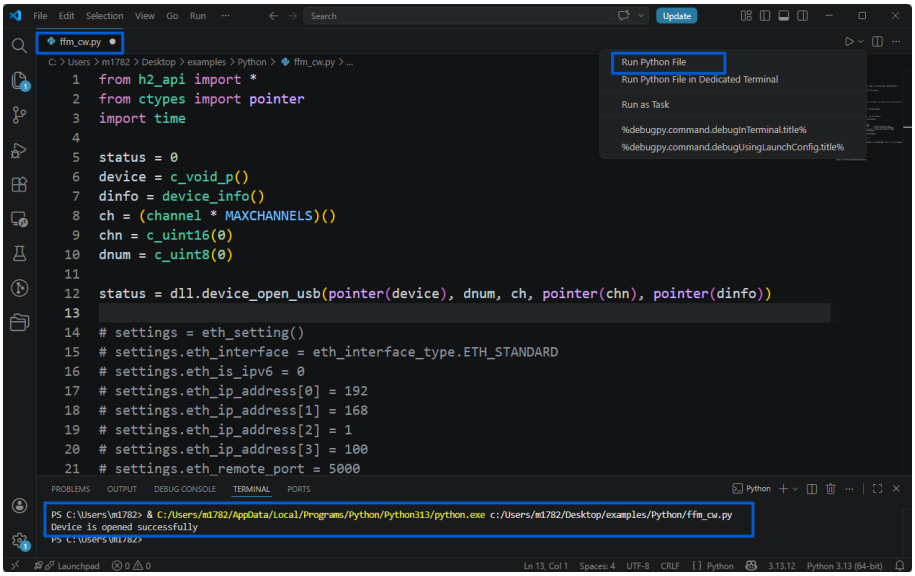

Figure 14 Run ffm_cw.py

2.4 C#

2.4.1 Configuring the Development Environment

Open Visual Studio Installer, select the “.NET Desktop Development” and “Universal Windows Platform Development” workloads, then click “Modify” to ensure that Visual Studio 2019 has a C# development environment.

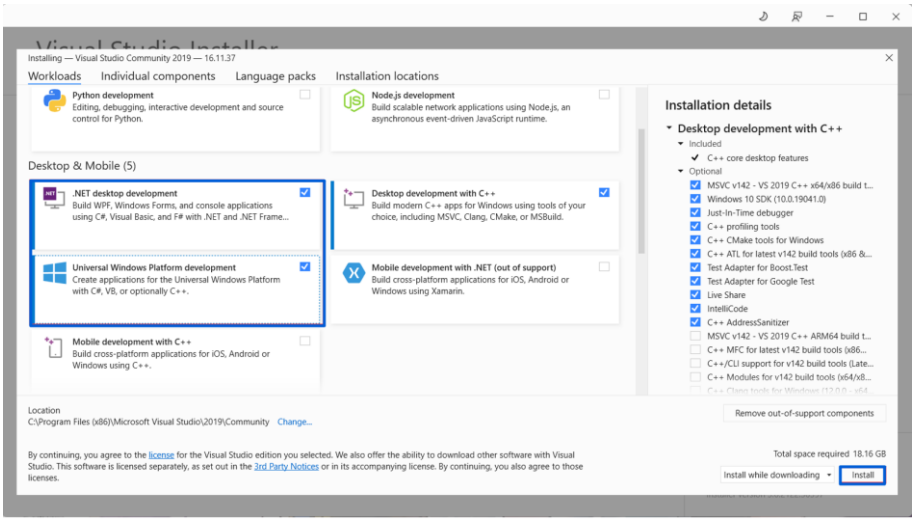

Figure 15 Configuring the C# Development Environment in Visual Studio

1. Open Visual Studio 2019 and click “Create a new project”;
2. Select the C# “Console App (.NET Framework)” and click “Next”;
3. Enter the project name (e.g., *ConsoleApp1*) and location. Uncheck “Place solution and project in the same directory”. Select “.NET Framework 4.5” as the framework, then click “Create”;
4. After creation, right-click the solution “ConsoleApp1” and select “Add” → “New Project”;
5. In the “Add New Project” dialog, choose the project type “Library”, then select the C# “Class Library (Universal Windows)”, and click “Next”;

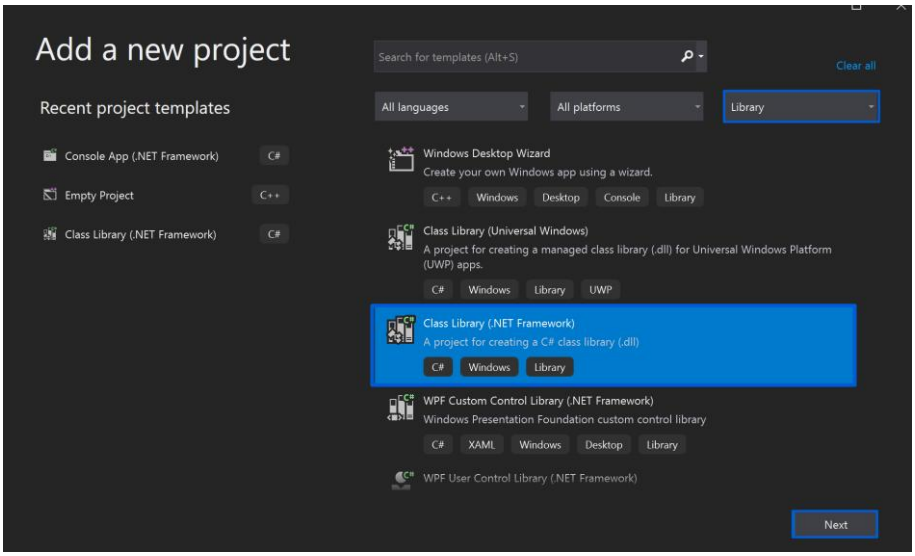

Figure 16 Add a New Project

6. Modify the project name as needed (e.g., *api*), keep the default location, select “.NET Framework 4.5” as the framework, and click “Create”;
7. Right-click *ConsoleApp1* and select “Properties”. In the project properties window, go to the “Build” tab and set the “Platform target” to “x86”. Then switch to the “Debug” tab, enter

“api\” in the “Working directory” field, click “OK” in the pop-up to confirm, and finally save and close the properties window;

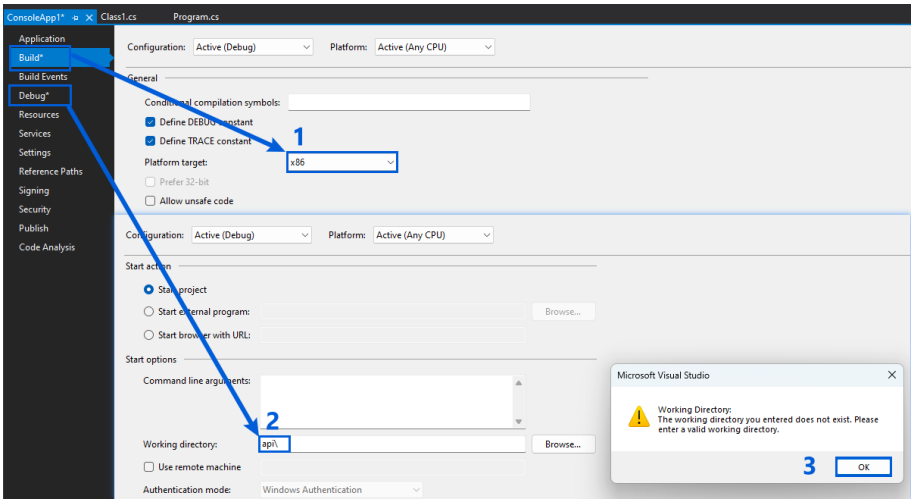

Figure 17 Configuring Project Properties

8. Right-click the class library “api” and select “Properties”. In the library properties window, go to “Build” and set the “Target Platform” to “x86”, then save;
9. Copy the contents of the file api.cs from the \Windows\examples\C# folder on the provided USB drive into the Class1.cs file of the project’s class library, replacing the existing code, and save it;
10. Select the ConsoleApp1 project, right-click “References”, choose “Add Reference”, check the “api” class library in the project list, click “OK”, and confirm to add the reference.

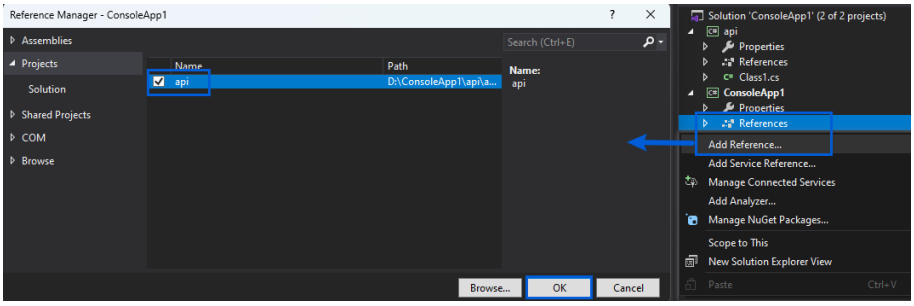

Figure 18 Add class library reference

11. Copy the contents from \Windows\api\x86 on the provided USB drive into the project folder’s bin\Debug directory, and ensure that the api\CalFile folder contains the calibration files for the instrument being used;

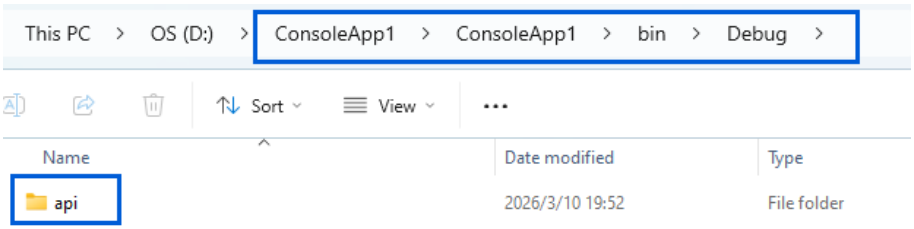

Figure 19 Copy API files

12. You may refer to the C# examples provided on the USB drive and write your code as needed.

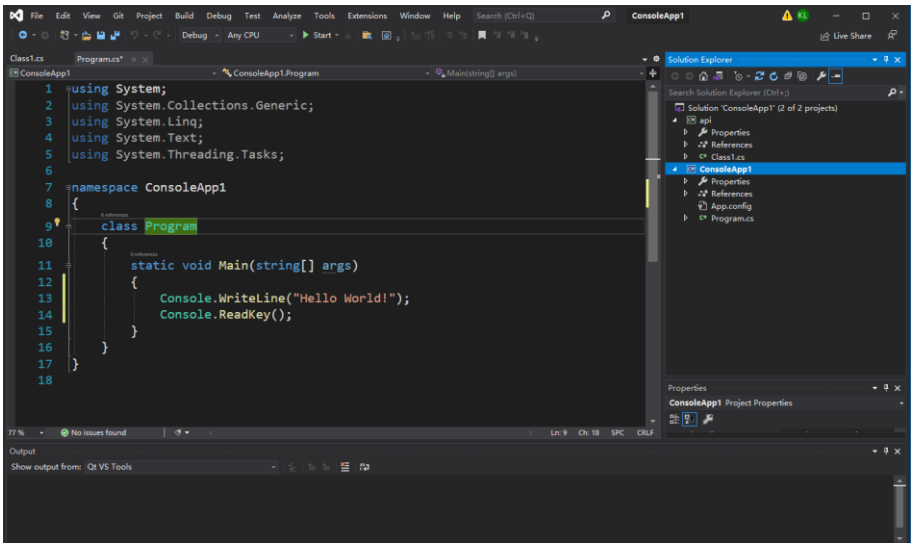

Figure 20 Write C# code

2.4.2 Example Usage Procedure

1. Open the solution CSharp_Examples.sln located in the Windows\examples\C# folder on the provided USB drive using Visual Studio.
2. Click the CSharp_Example project on the right side, then open the Programe.cs file within it.

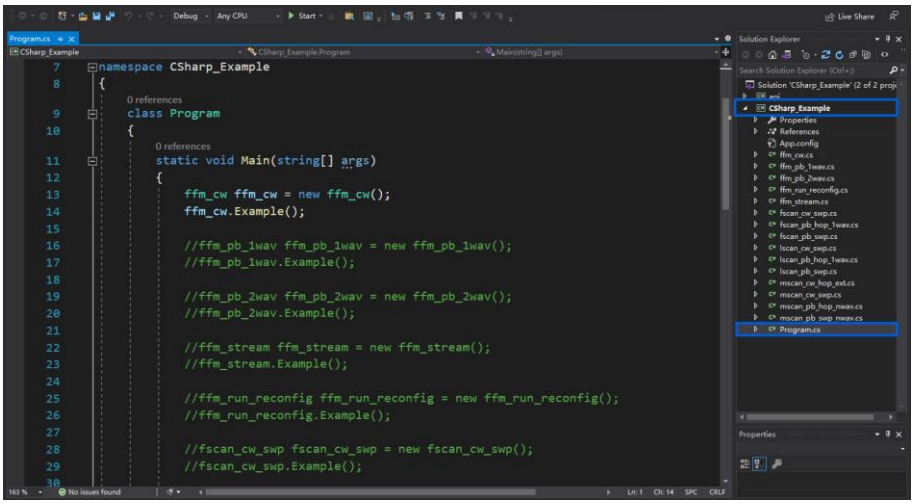

Figure 21 Open the Programe.cs file

3. Each example in the provided C# sample is encapsulated in a separate class. To use it, simply uncomment the corresponding code (multiple examples cannot be used simultaneously).

2.5 Matlab

The following section uses MATLAB 2016a as an example to describe how to call the 64-bit htra_api dynamic link library.

2.5.1 Example Usage Procedure

1. Install a C++ compiler.
2. Properly connect the device, open MATLAB, and copy the path of the "Windows\Examples\Matlab" folder from the shipped USB flash drive into the MATLAB Current Folder address bar, then press Enter.
3. Click the "ffm_cw.m" file on the left side, and verify that the compiler path in the first line of the example matches the locally installed compiler path.

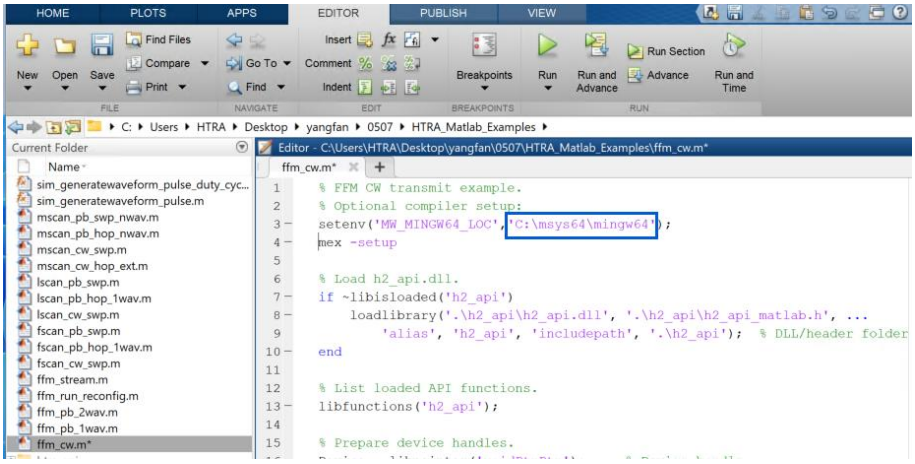

Figure 22 Verifying Compiler Path

4. Click "Run". A successful program execution is illustrated below. For the function description of each example, refer to the Example Description section.

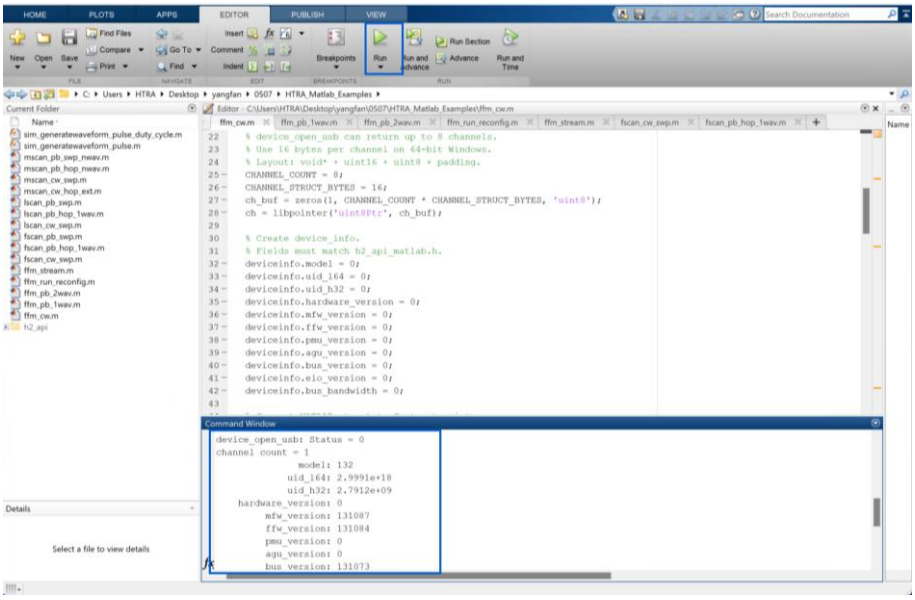

Figure 23 Running the ffm_cw.m File

Note: Since calling dynamic link libraries requires a C++ compiler, it is recommended to install the g++ compiler using MSYS2. Refer to https://www.msys2.org/ for download and installation.

3. Linux

3.1 System Operating Environment Check

When using the device on a Linux system, first verify whether the current system architecture, GCC version, and GLIBC version are supported by following the steps below:

1. Open a terminal and enter "uname -a" to check the Linux system architecture (in the example below, the system architecture is x86_64);
2. Enter "gcc -v" to check the system GCC version (in the example below, the GCC version is 7.5.0);
3. Enter "ldd --version" in the terminal to check the system GLIBC version (in the example below, the GLIBC version is 2.27);
4. Compare the information obtained from the terminal with the reference table to verify whether the current environment is supported. If it is not supported, please contact technical support.

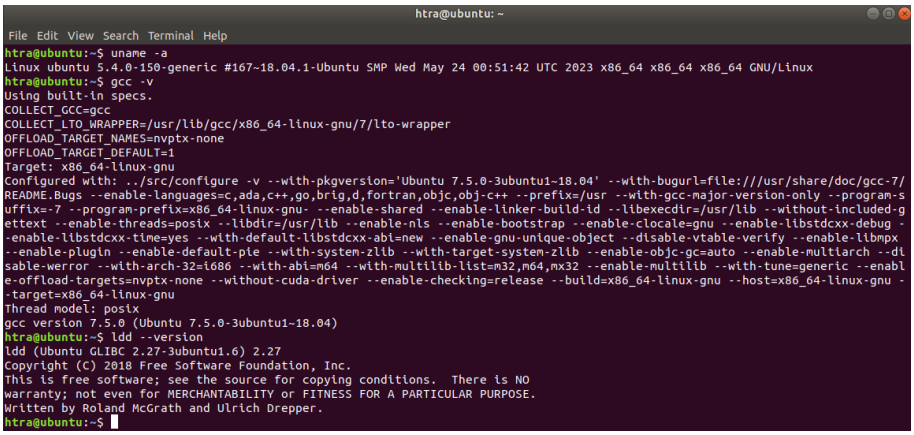

Figure 24 Checking Linux System Architecture and Version

Table 1 Terminal Information Comparison Table

x86 Processor	Supports Intel and AMD processors
ARM Processor	aarch64 (ARMv8) and armv7 processors, eg, Raspberry Pi 4B, RK3399, RK3568, RK3588, T507, NVIDIA Jetson TX2
Build Environment	GCC 4.8, GLIBC 2.17 or later
Distribution	Raspberry Pi 4B custom system, Ubuntu 18.04, etc.

3.2 Included Documentation Description

The Linux package provided on the shipped USB flash drive includes the following materials:

3.2.1 C++ Example

The "\Linux\examples" directory contains the following items in the C++ folder:

1. demo folder: C++ example programs (for usage instructions, refer to the C++ Example Usage section).
2. Makefile file: A build script used to compile the example programs into executable files.
3. bin folder: Used to store parameter restriction files and the executable files generated after building the example programs.

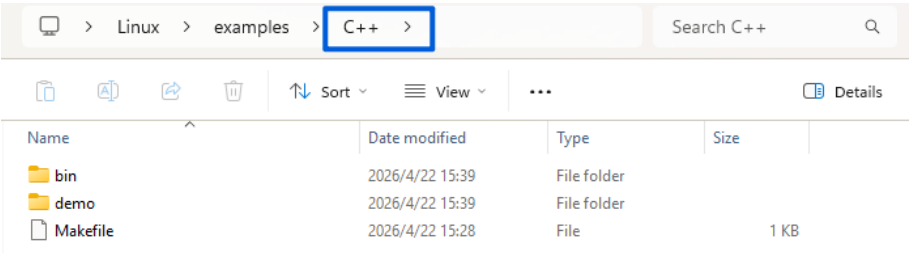

Figure 25 Contents of the C++ Folder under Linux

3.2.2 Qt Example

The "\Linux\examples" directory contains the following items in the Qt folder:

1. demo folder: Qt example applications and project (*.pro) files (for usage instructions, refer to the Qt Example Usage section).
2. bin folder: Used to store device parameter restriction files and executable files generated after compiling the example programs.
3. api folder: Used to store dynamic link libraries.

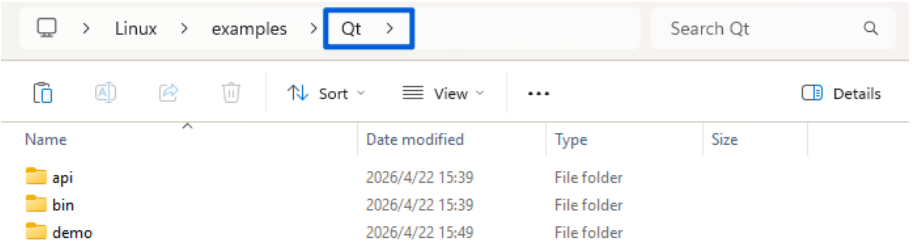

Figure 26 Contents of the Qt Folder under Linux

3.2.3 Python Example

The "\Linux\examples" directory contains the following items in the Python folder:

1. Python example programs (for usage instructions, refer to the Python Example Usage section).
2. CalFile folder: Used to store device parameter restriction files.
3. api folder: Used to store dynamic link libraries.

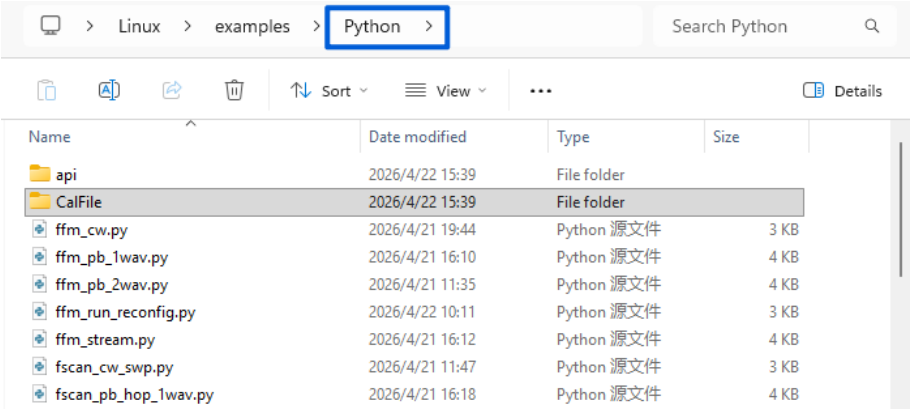

Figure 27 Contents of the Python Folder under Linux

3.2.4 Install_H2_SDK

1. The Install_H2_SDK folder contains:

   install_h2_lib.sh: Driver configuration script

   h2api folder: Stores drivers, header files, and libraries

2. The Install_H2_SDK/h2api folder contains:

   Configs folder: Driver configuration files

   Inc folder: Header files

   Lib folder: Dynamic link libraries

3. The Install_H2_SDK/h2api/lib folder contains:

   aarch64 folder: Dynamic link libraries for the aarch64 architecture

   x86_64 folder: Dynamic link libraries for the x86_64 architecture

   armv7 folder: Dynamic link libraries for the armv7 architecture

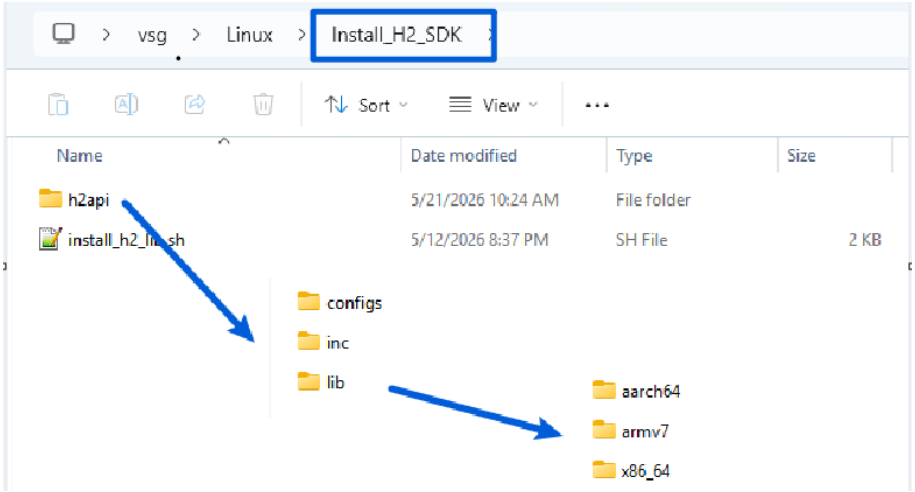

Figure 28 Contents of the Install_H2_SDK Folder under Linux

3.3 Driver Configuration

Before using the device on Linux, the driver must be configured. The procedure is as follows:

1. Copy the Install_H2_SDK folder to the Linux host machine. Open a terminal in the Install_H2_SDK directory and execute "sudo sh install_h2api_lib.sh" to configure the driver files.

   Note: If the development board does not support the sudo command, run: "sh install_h2api_lib.sh"

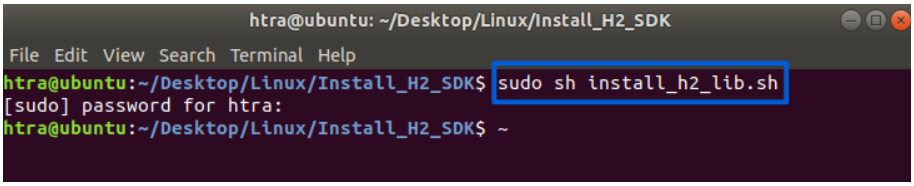

Figure 29 Driver Configuration under Linux

2. Properly connect the device to the Linux host computer and ensure that the device is correctly powered. If the host is running in a virtual machine, the device must be connected to the virtual machine first, and USB compatibility should be set to USB 3.1. In the terminal, enter "lsusb" to view the list of USB devices. If "ID: 6430" (or "ID: 04b5" or “ID: 367f") appears, the device has been successfully detected.

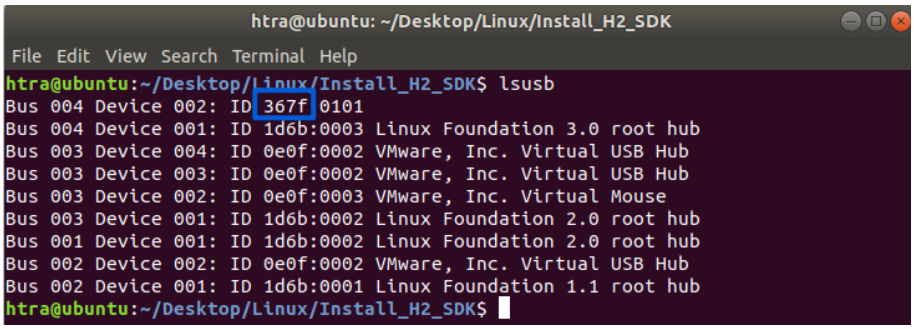

Figure 30 Checking Device Connection Status under Linux

3.4 C/C++

3.4.1 Example Usage Procedure

Prerequisites: Ensure that the device is properly connected and that the driver has been correctly configured according to the Driver Configuration section.

The following procedure describes how to use the C++ examples provided on the shipped USB flash drive (for detailed descriptions of each example, refer to the Example Description section):

1. Select the program to be compiled: Copy the "Linux\examples\C++" folder from the

shipped USB flash drive to the host computer. Open "main.cpp" in the demo folder and uncomment the example that you want to test.

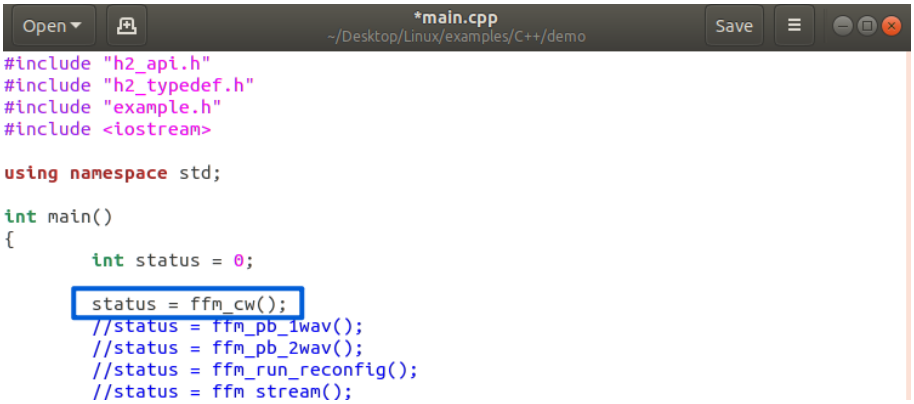

Figure 31 Uncommenting the ffm_cw Example

2. Compile the selected example: Refer to the System Operating Environment section to determine the system architecture. Open a terminal in the C++ folder and enter the corresponding command according to the host system architecture (the ffm_cw example is used below as an example):

   For the x86_64 architecture: make Example=ffm_cw

   For the aarch64 architecture: make TARG=aarch64 Example=ffm_cw

   For the armv7 architecture: make TARG=armv7 Example=ffm_cw

3. Run the program: After the program has been successfully compiled, open a terminal and enter "./bin/ffm_cw " to run the selected example.

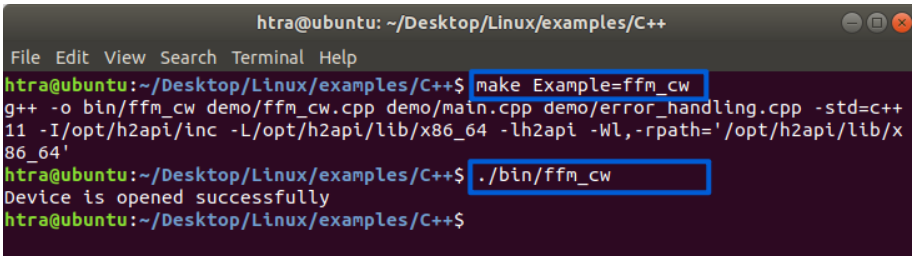

Figure 32 Compiling and Running the ffm_cw Example on x86_64 Architecture

3.4.2 Creating and Compiling a New Project

Prerequisites: Ensure that the device is properly connected and that the driver files have been correctly configured according to the Driver Configuration section.

1. Code Development:

The Linux dynamic link libraries provided on the shipped USB flash drive are fully consistent with those used on Windows. Code development can therefore follow the logic and conventions described in the API Programming Guide.

2. Compile and Run:

(1) Create a new folder for the project (for example, C++_Test). Then create a CalFile folder within the project directory to store parameter restriction files, and create an h2apifolder to store header files and dynamic link libraries.

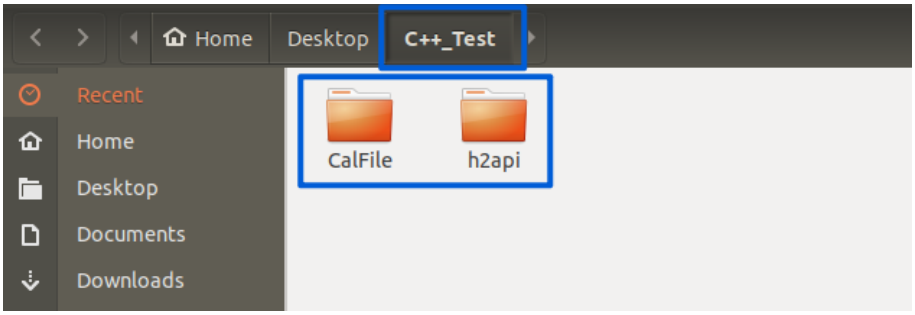

Figure 33 Creating the Project Folder

(2) Create an inc folder under the h2apidirectory for storing header files, and a lib folder for storing dynamic link libraries.

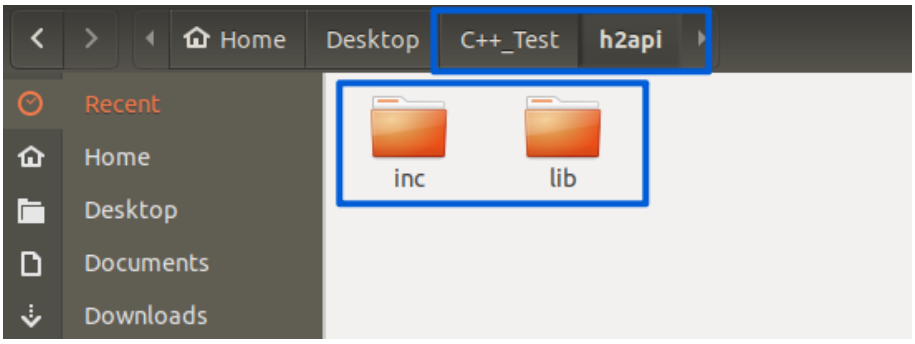

Figure 34 Creating inc and lib Folders under h2api

(3) Copy the files from the CalFile folder on the shipped USB flash drive to the newly created "C++_Test\CalFile" folder.

(4) Copy the header files from "Linux\Install_H2_SDK\h2api\inc" on the shipped USB flash drive to the newly created "C++_Test\h2api\inc" folder.

(5) Refer to the System Operating Environment section to determine the system architecture. According to the description of the lib folder, copy the files from the corresponding architecture folder into the "C++_Test\h2api\lib folder".

(6) Open a terminal in the lib folder and enter the following command to create symbolic links for the copied dynamic link libraries (the symbolic link command is the same for all three architectures):

The libh2api.solibrary below uses version 2.0.17 as an example. For other versions, replace the version number accordingly.

```
ln -sf libh2api.so.2.0.17 libh2api.so.2
ln -sf libh2api.so.2 libh2api.so
ln -sf libusb-1.0.so.0.2.0 libusb-1.0.so.0
ln -sf libusb-1.0.so.0 libusb-1.0.so
```

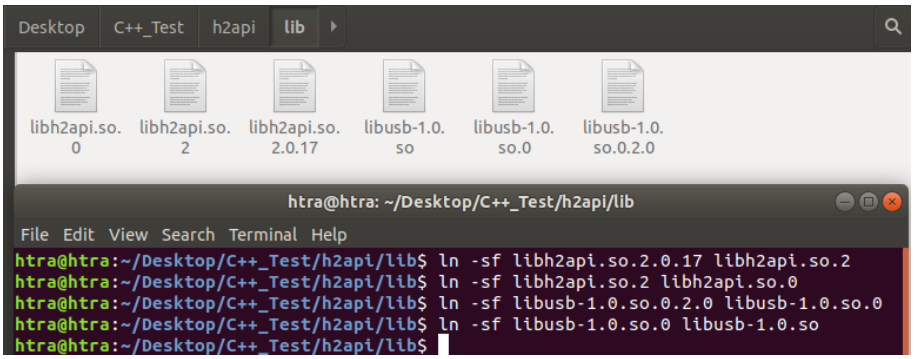

Figure 35 Creating Symbolic Links for Dynamic Link Libraries

(7) Place the developed source code files in the root directory of C++_Test.

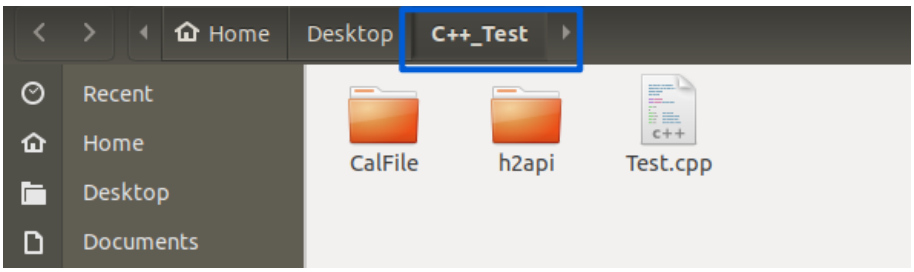

Figure 36 Placing Source Code Files

(8) Compile and generate the executable file: According to the system architecture confirmed in the System Operating Environment section, open a terminal in the C++_Test folder and enter the corresponding compilation command.

The following example uses Test.cpp to compile and generate an executable file named Teston different system architectures:

x86_64 architecture:

g++ -o Test Test.cpp -std=c++11 -I ./h2api/inc -L ./h2api/lib -lh2api -Wl,-rpath='./h2api/lib'

aarch64 architecture:

aarch64-linux-gnu-g++ -o Test Test.cpp -std=c++11 -I ./h2api/inc -L ./h2api/lib -lh2api -Wl,-rpath='./h2api/lib'

armv7 architecture:

arm-linux-gnueabihf-g++ -o Test Test.cpp -std=c++11 -I ./h2api/inc -L ./h2api/lib -lh2api -Wl,-rpath='./h2api/lib'

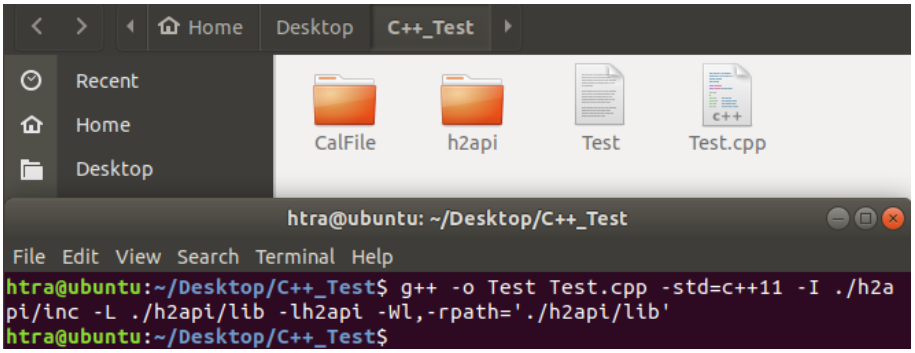

Figure 37 Compiling to Generate the Executable File

(9) Run the program: Enter "./Test" to start the executable file.

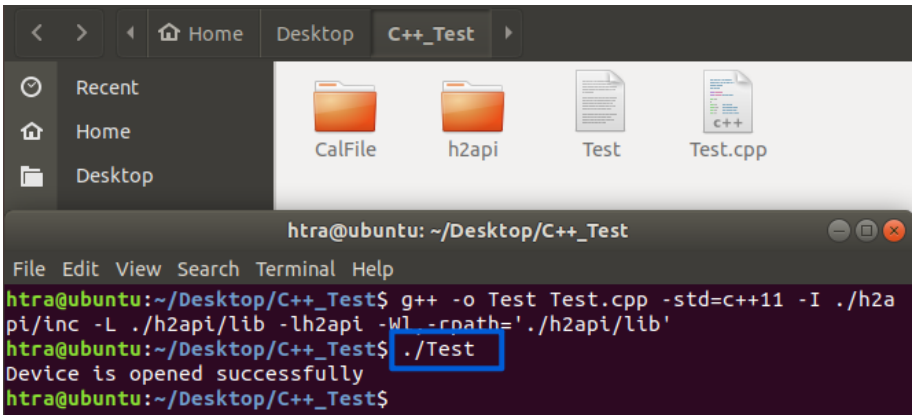

Figure 38 Running the Test Executable File

3.4.3 Cross Compilation

If a cross-compilation toolchain is available on the host computer, follow the procedure below to cross-compile applications for the device.

The following example describes cross-compiling an aarch64 executable on an x86_64 host system:

1. Generate the target architecture executable file:

(1) Follow steps (1)–(8) in the Creating and Compiling a New Project section to create the project, and place the header files and library files for the target architecture (aarch64). Then perform symbolic linking, write the program, and compile the executable using the compiler corresponding to the target architecture. As shown below, when compiling an aarch64 executable, the aarch64 toolchain compiler command should be used.

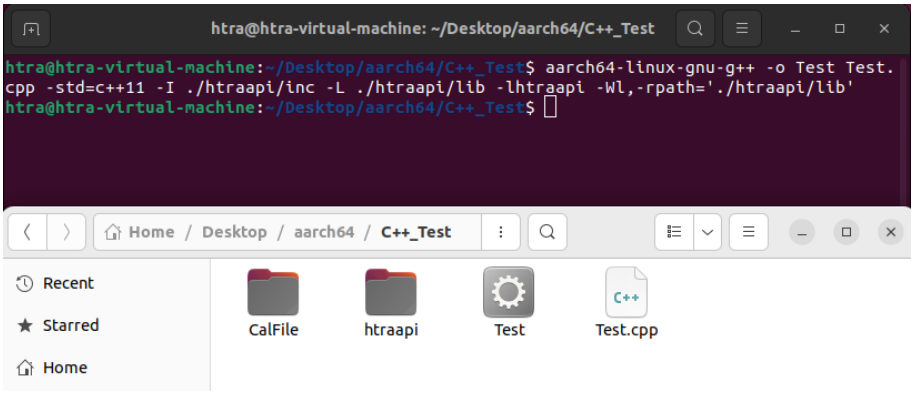

Figure 39 Cross-Compiling an Executable for the aarch64 Architecture

(2) After generating the executable file, enter "file Test " to check the architecture of the executable. It can be seen that the current executable file architecture is aarch64.

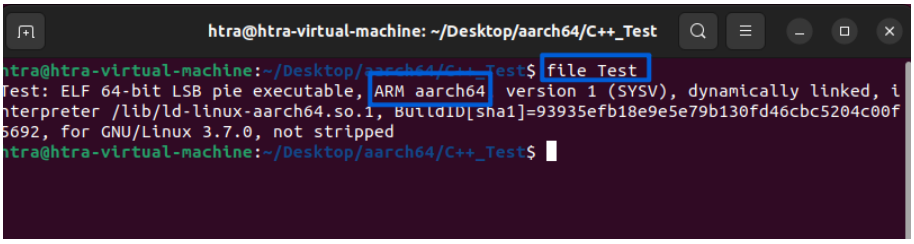

Figure 40 Checking the Executable File Architecture

2. Run the executable on an aarch64 host machine:

(1) Navigate to the project directory and execute "zip -r C++_Test.zip C++_Test " to compress the entire C++_Test folder into a ZIP archive. Then copy the generated archive to the aarch64 host machine.

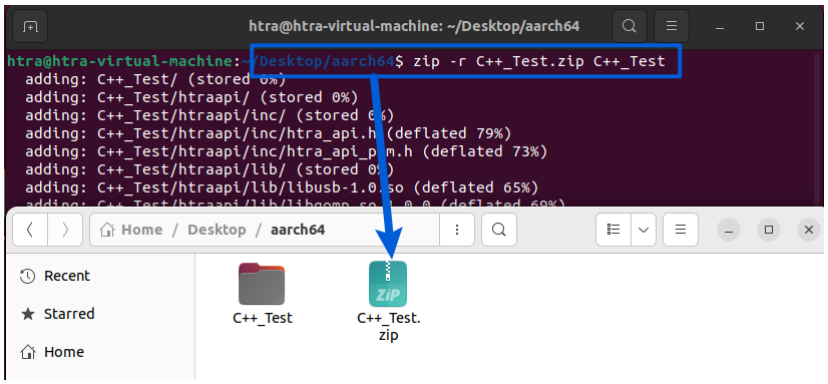

Figure 41 Compressing the Created Project

(2) On the aarch64 host, enter "unzip C++_Test.zip " to extract the project.

(3) Refer to the Driver Configuration section and configure the driver files on the aarch64 host machine.

(4) After the driver configuration is completed, enter the C++_Test directory and run the program by executing "./Test" in the terminal.

3.5 QT

3.5.1 Example Usage Procedure

Prerequisites: Ensure that the device is properly connected and that the driver has been correctly configured according to the Driver Configuration.

The following example describes running the Qt example on Ubuntu 18.04 with an x86_64 architecture system.

1. Copy the "Linux\examples\Qt " folder from the shipped USB flash drive to the host machine.
2. Refer to the System Operating Environment Check to check the system architecture, and copy the corresponding architecture folder contents from "Linux\Install_H2_SDK\h2api\lib " on the shipped USB flash drive into the "Qt\api " folder on the host machine.

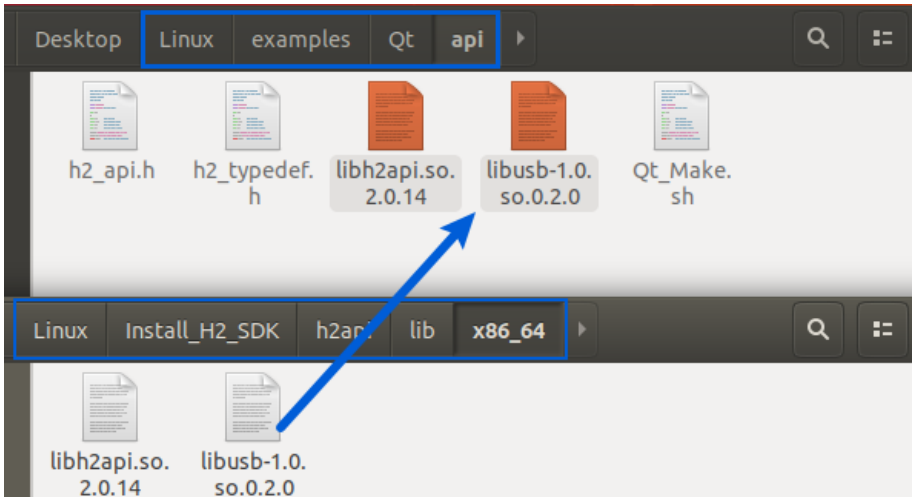

Figure 42 Copying Dynamic Link Libraries for the Corresponding Architecture

3. Open a terminal in the api folder and enter "sudo sh Qt_Make.sh", then follow the prompts to enter the password to authorize creation of symbolic links.

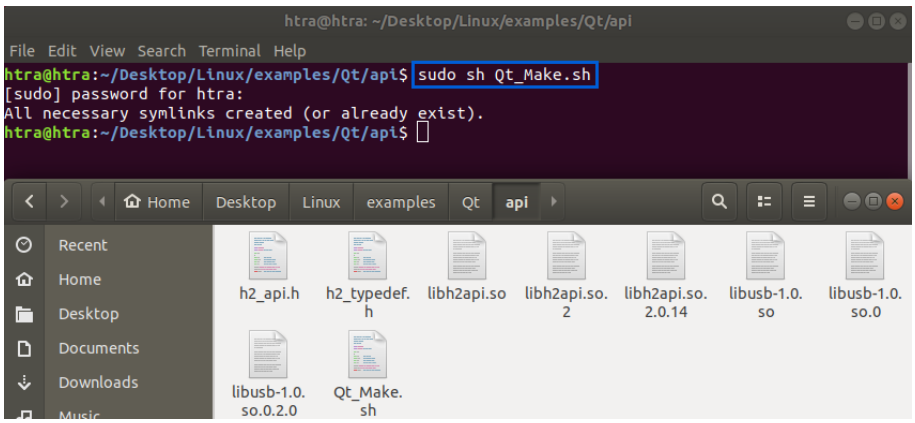

Figure 43 Creating Symbolic Links for Dynamic Link Libraries

4. Open the demo.profile in "Qt\demo" using Qt Creator, and configure the project build environment.

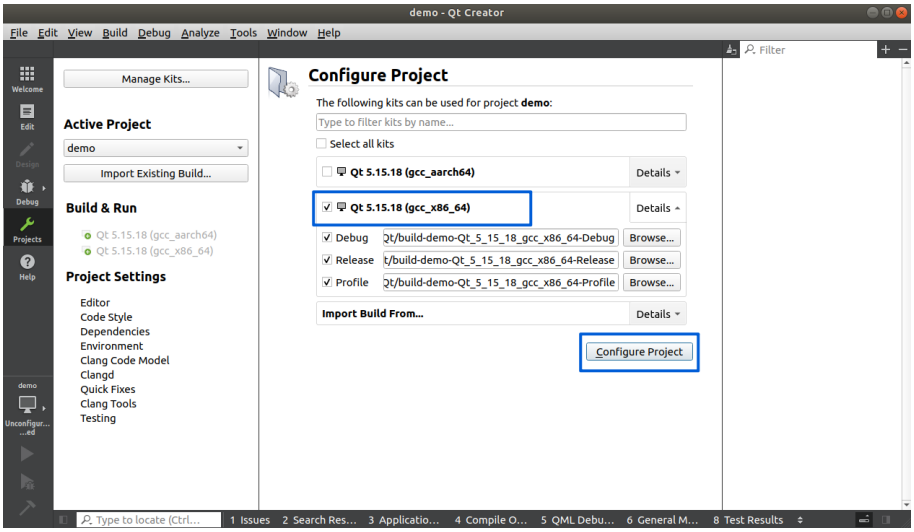

Figure 44 Build Environment

5. Click "Edit", open "main.cpp" under the Sources folder in the "htrademo" project, uncomment the relevant functions, save the file, and click Run to execute different examples.

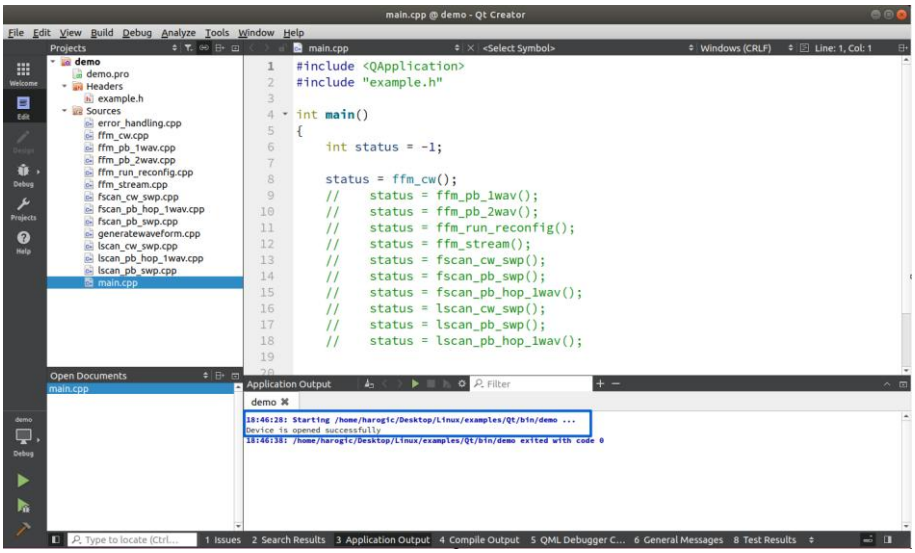

Figure 45 Running the ffm_cw Example

3.5.2 Creating and Compiling a New Project

Under the premise that the driver files have been correctly configured according to the Driver Configuration section, follow the procedure below to create and compile a Qt project.

The code implementation shall follow the API Programming Guide. The procedure for creating a GUI application is as follows:

1. Create a new folder for the entire project (eg, QtTest), which includes a bin folder (used to store calibration files and generated executable files) and an h2apifolder (used to store header files and dynamic link libraries).
2. Copy the "CalFile" folder from the instrument's shipped USB flash drive into the QtTest\bin folder.
3. Copy the header files from "Linux\Install_H2_SDK\h2api\inc" on the shipped USB flash drive into the "QtTest\h2apifolder".
4. Copy the dynamic link libraries from the corresponding system architecture folder under "Linux\Install_H2_SDK\h2api\lib" on the shipped USB flash drive into the "QtTest\h2apifolder" (x86_64 architecture is used as an example here).

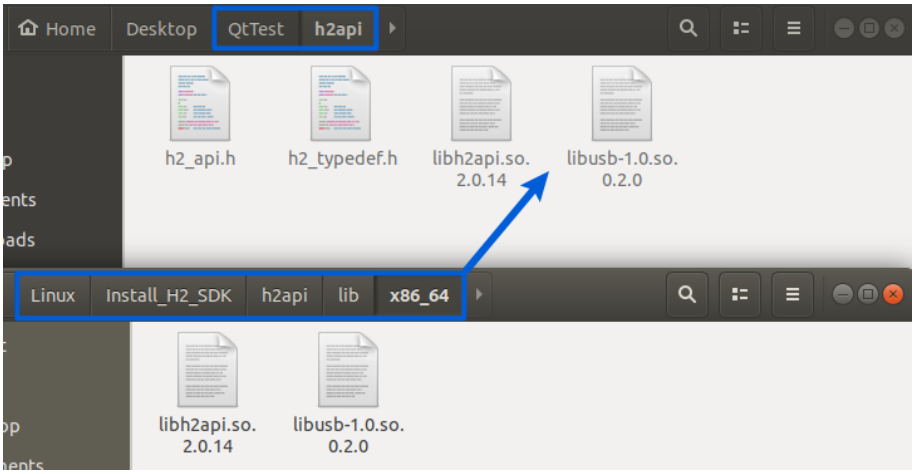

Figure 46 Copying Dynamic Link Libraries for the Corresponding Architecture

5. Open a terminal in the h2apifolder and enter the following commands sequentially to create symbolic links for the copied dynamic link libraries (the commands are identical across different architectures).

The libh2api.solibrary below uses version 2.0.14 as an example. For other versions, replace the version number accordingly.

```
ln -sf libh2api.so.2.0.14 libh2api.so.2
ln -sf libh2api.so.2 libh2api.so
ln -sf libusb-1.0.so.0.2.0 libusb-1.0.so.0
ln -sf libusb-1.0.so.0 libusb-1.0.so
```

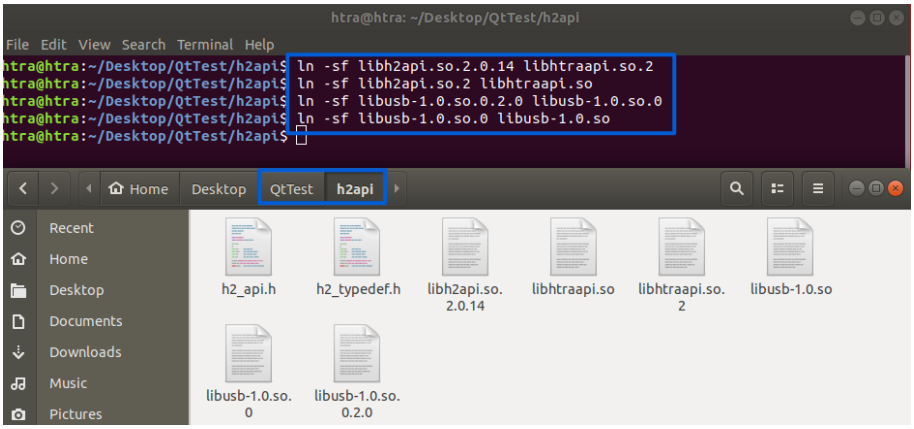

Figure 47 Creating Symbolic Links for Dynamic Link Libraries in h2api

6. Open Qt Creator and click "File" > "New File or Project"
7. In the project section, select "Application", then choose "Qt Widgets Application" and click "Choose...",
8. Enter the project name (eg, test), click "Browse", select the previously created QtTest folder, and click "Next".
9. Select "qmake" and continue clicking "Next" until the "Kit Selection" interface appears. Choose a build kit and click "Next".
10. Click "Finish" to create the project.
11. In the Qt Creator main interface, right-click the "test" project and select "Add Library..." > "External Library" > "Next".
12. Click Browse Library File, select "libh2api.so" in the "QtTest\h2api" folder, and click "Open" . Select the "Linux" platform and click "Next".

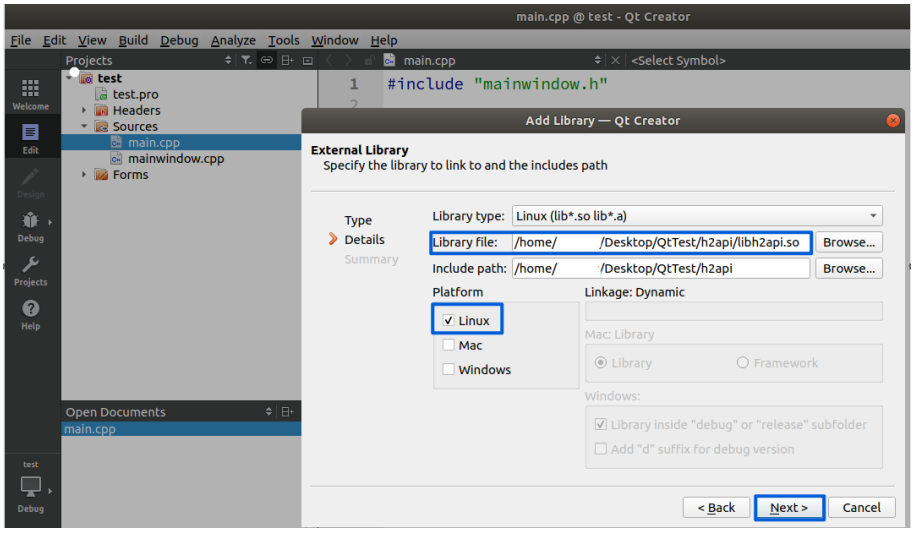

Figure 48 Selecting the Linux Platform

13. Click "Finish" to add the external library.
14. In "test.pro", after "CONFIG += c++11", add "DESTDIR = $$clean_path($$PWD/../bin)" to specify the output directory of the executable file.

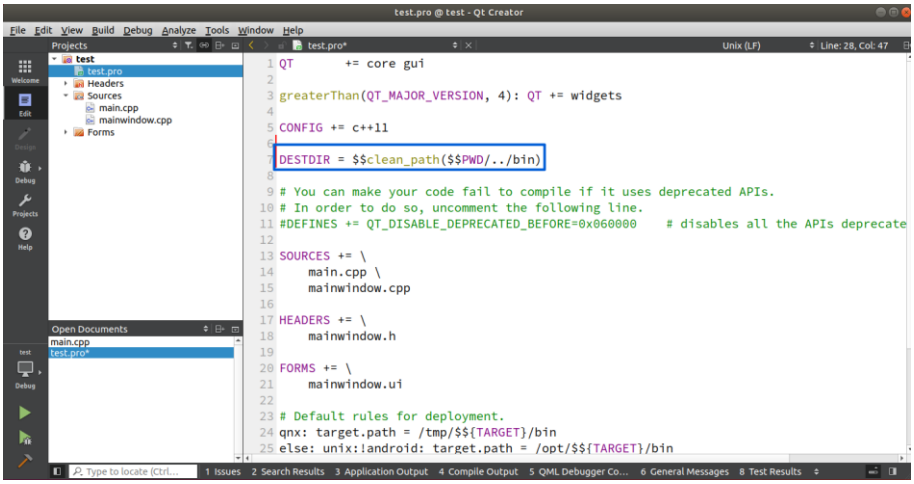

Figure 49 Setting the Executable Output Path

15. After the library configuration, add "-Wl,-rpath,$$PWD/../h2api" to specify the runtime library search path.

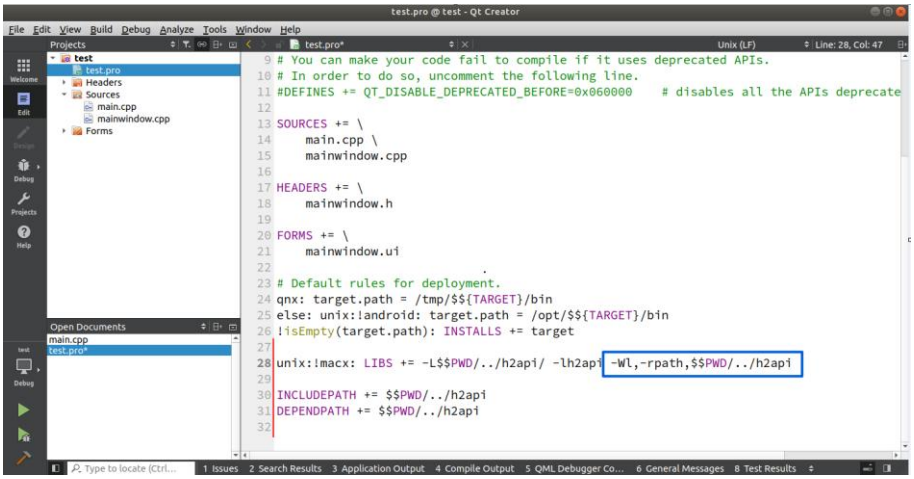

Figure 50 Specifying the Library Link Path for the Executable

16. Save the "test.profile", then write code in "mainwindow.cpp" and click "Run". The normal running interface is shown below.

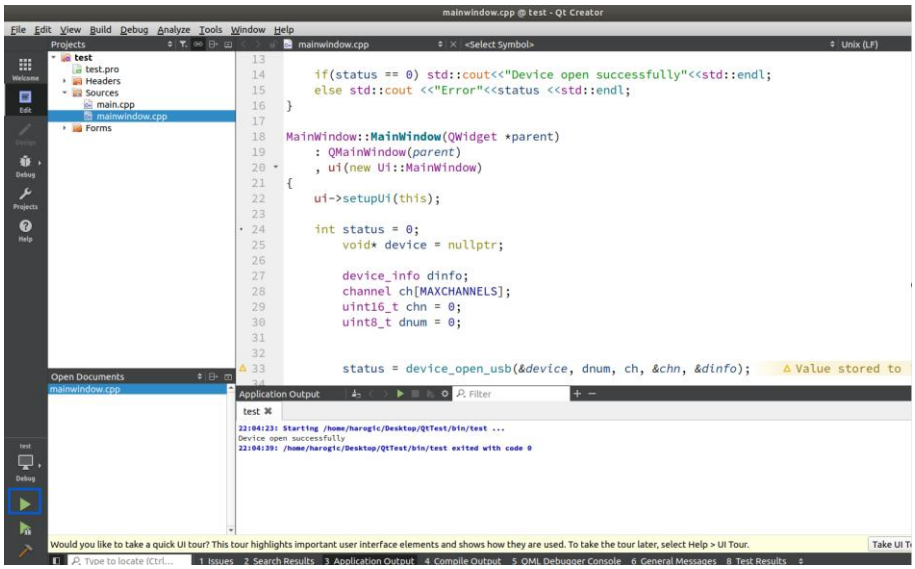

Figure 51 Running the Code

17. Close QtCreator, navigate to the "QtTest\bin" folder, open a terminal, and enter "./test" to run the executable program.

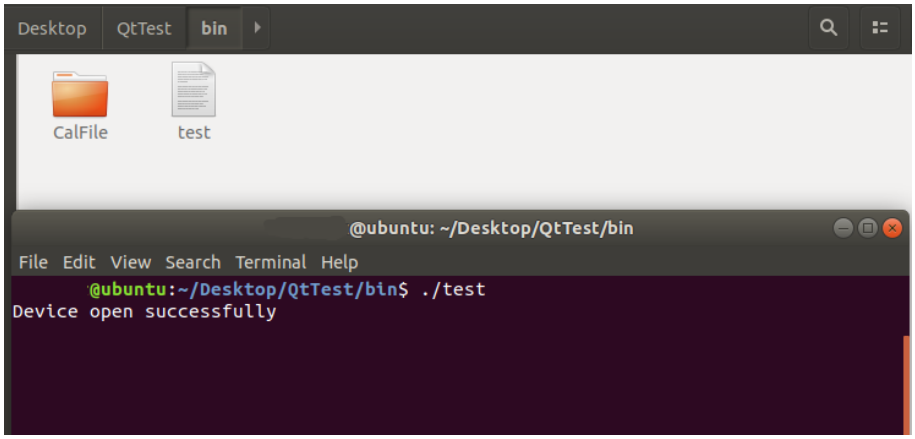

Figure 52 Running the Executable File

3.5.3 Cross Compilation

If a cross-compilation toolchain is available on the host computer, follow the procedure below to cross-compile applications for the device (the following example describes cross-compiling an aarch64 executable on an x86_64 host system).

1. Generate the target architecture executable file:

(1) Follow steps 1–9 in the Qt GUI Application Creation Procedure described in the Creating and Compiling a New Project section to create the project. Place the parameter restriction files, header files, and dynamic link libraries for the target cross-compilation architecture (aarch64), configure symbolic links for the dynamic libraries, create the project in Qt Creator, and select the aarch64 build kit.

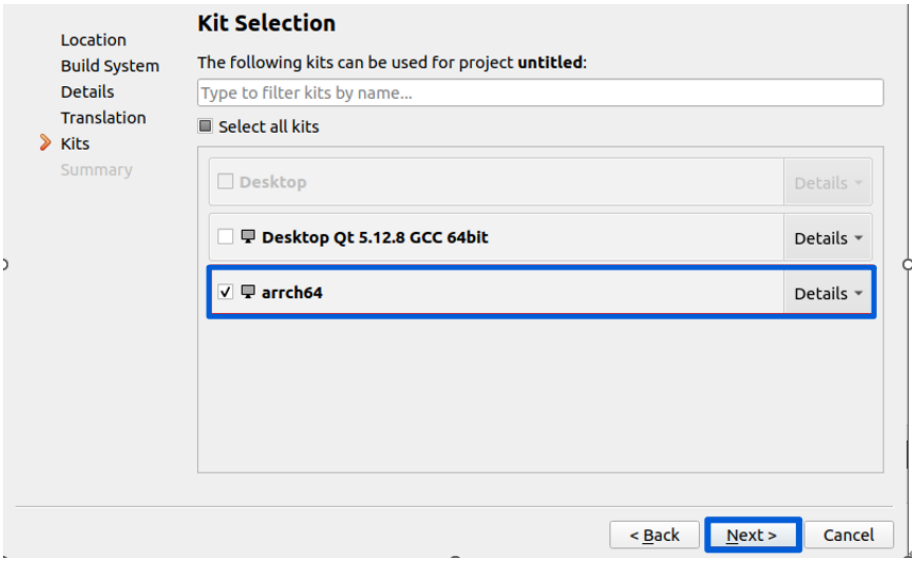

Figure 53 Selecting the Target Kit for Cross-Compilation

(2) Follow steps 10–16 of the GUI application creation procedure to complete project creation, library integration, executable output path configuration, code implementation, and execution.

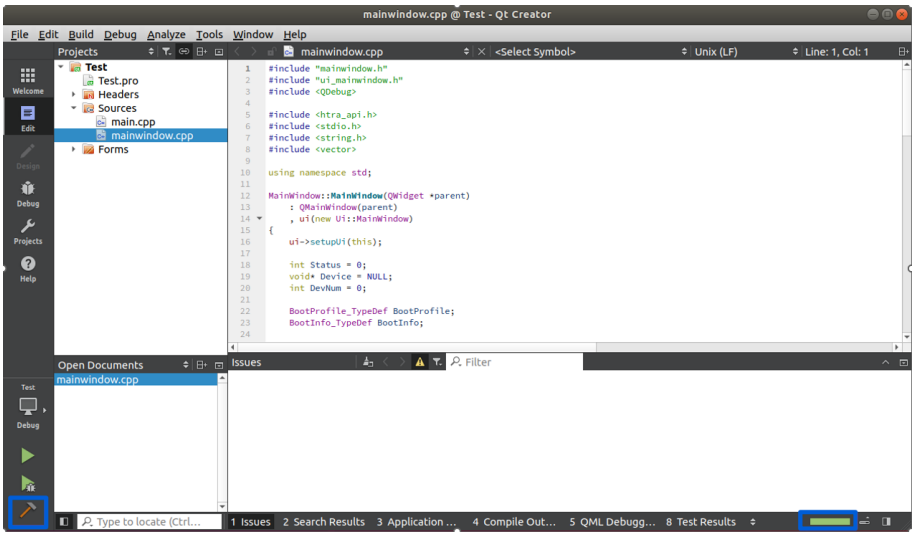

Figure 54 Building the Executable for the aarch64 Architecture

(3) After building the executable, open a terminal in the "QtTest\bin" folder and enter "file Test" to check the architecture of the generated executable (in this example, the executable name is Test).

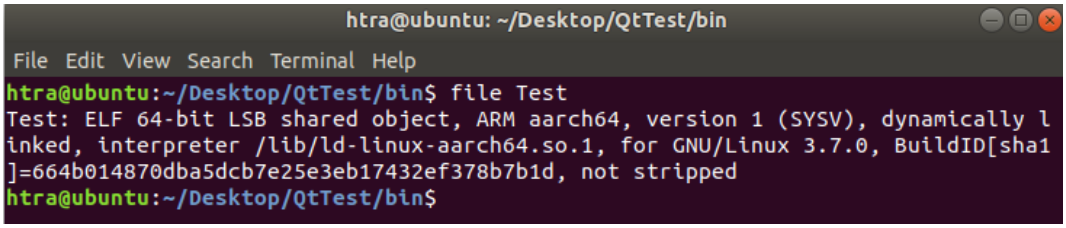

Figure 55 Checking the Executable File Architecture

2. Run the executable on an aarch64 host machine:

(1) Navigate to the project directory (in this example, the project is located on the desktop, so use "cd Desktop/"). Execute "zip -r QtTest.zip QtTest" to compress the entire QtTest folder into a ZIP archive, then copy the archive to the aarch64 host machine.

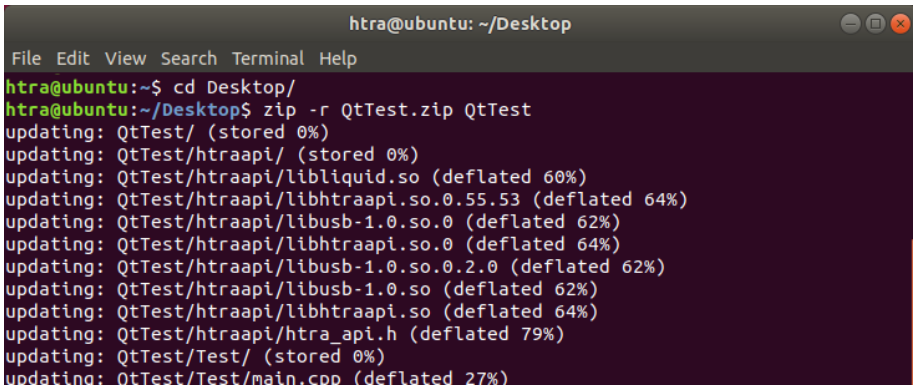

Figure 56 Compressing the Created Project

(2) On the aarch64 host, enter "unzip QtTest.zip" to extract the project.

(3) Refer to the Driver Configuration section to configure the driver files on the aarch64 host machine.

(4) After driver configuration is completed, enter the QtTest folder, run "chmod 777" Testin the terminal to grant execute permissions, and then execute "./Test" to run the program.

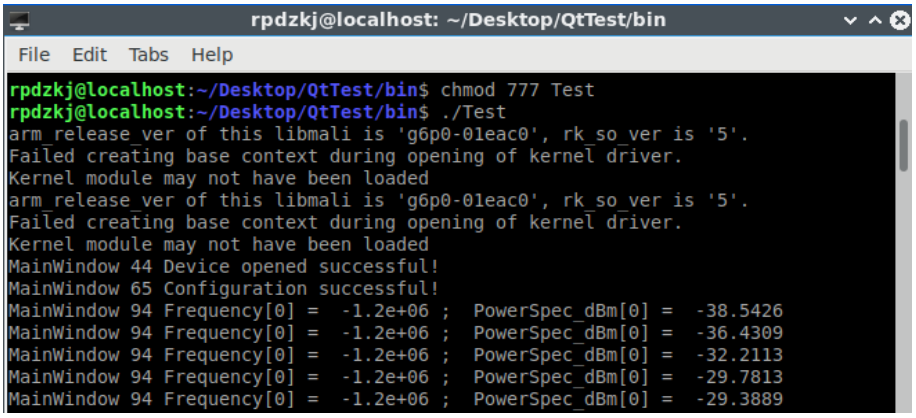

Figure 57 Granting Permissions and Running the Executable File

3.6 Python

3.6.1 Example Usage Procedure

Prerequisites: Ensure that the device is properly connected and that the driver files have been correctly configured according to the Driver Configuration section.

1. Copy the "Linux\examples\Python" folder from the shipped USB flash drive to the host machine. Open a terminal in this folder and enter "which python3" to check the location of the Python interpreter (for example, the location shown here is /usr/bin).

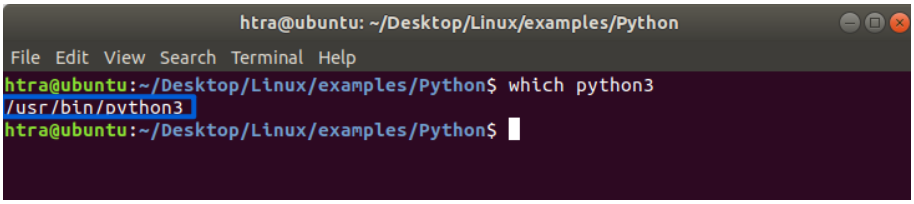

Figure 58 Checking Python 3 Interpreter Path

2. According to the obtained interpreter path, enter "sudo cp -r CalFile /usr/bin" to copy the CalFile folder to the same directory level as the Python interpreter (modify the copy path according to the actual interpreter location).

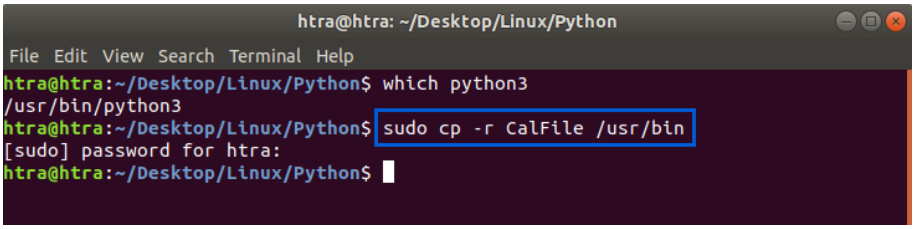

Figure 59 Copying the CalFile Folder to the Same Directory Level as the Interpreter

3. Refer to the System Operating Environment section to determine the system architecture, and copy the contents of the corresponding architecture folder from "Linux\Install_H2_SDK\h2api\lib" on the shipped USB flash drive to the "Python\api" folder on the host machine.

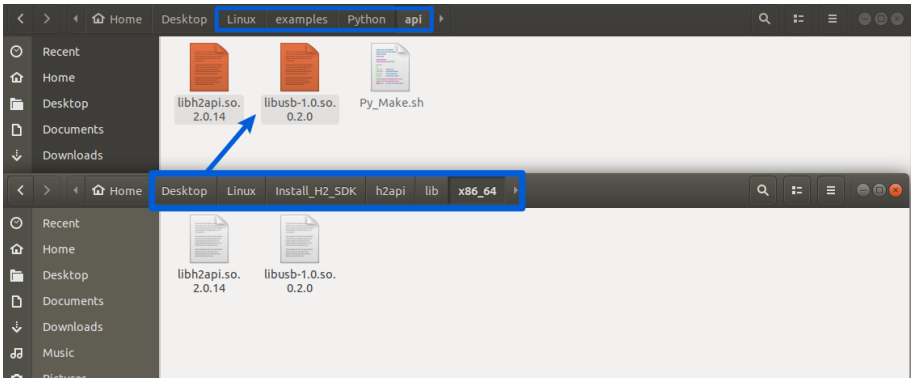

Figure 60 Copying Dynamic Link Libraries

4. Open a terminal in the apifolder and enter "sudo sh Py_Make.sh". Follow the prompts to enter the password and grant permission to create symbolic links for the libraries.

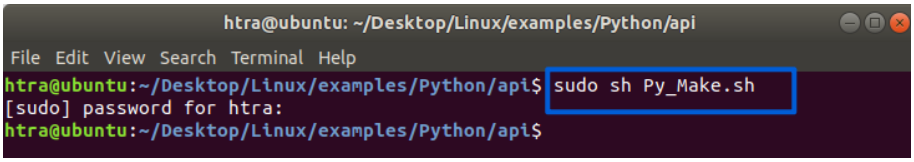

Figure 61 Creating Symbolic Links for the Libraries

5. Open a terminal in the Python folder and enter "python3 ffm_cw.py" to run the provided ffm_cw.py example.

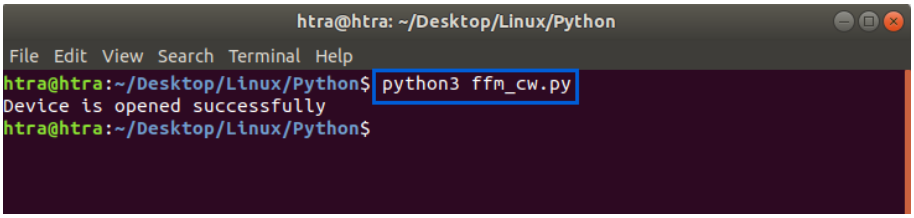

Figure 62 Running the Python Example

3.6.2 Creating and Running a New Project

Under the premise that the driver files have been correctly configured according to the Driver Configuration section, follow the procedure below to create a Python project:

1. The code implementation shall follow the API Programming Guide.
2. When running the program, place the script in the example folder (Python) and follow the procedure described in the Python Example Usage section.

4. Example Description

4.1 Continuous Wave Transmission

ffm_cw: Outputs a continuous wave at a single frequency point without involving waveform files. Suitable for frequency point verification and RF link connectivity testing.

4.2 Play Single Waveform File

ffm_pb_1wav: Plays back a single IQ waveform file at a single frequency point. Parameters such as waveform repeat count can be configured. Suitable for single modulation signal verification and demodulation testing.

4.3 Streaming

ffm_stream: Transmits IQ data in real time at a fixed frequency point, suitable for large data volume scenarios.

4.4 Dynamic Configuration Update

ffm_run_reconfig: Changes configuration during device operation, suitable for dynamic frequency/power adjustment and automated testing.

4.5 Frequency Sweep

fscan_cw_swp: Continuous wave automatic frequency sweep. Upon receiving a trigger, frequency sweeping is performed automatically according to the configured step size and dwell time.

fscan_pb_swp: Automatic modulated signal frequency sweep. Upon receiving a trigger, frequency is switched automatically according to the configured step size and dwell time, and the preloaded baseband waveform stored in internal memory is continuously played at each frequency point.

fscan_pb_hop_1wav: Single-step modulated signal frequency hopping. Frequency changes step-by-step based on external commands, and the preloaded waveform is continuously played at the current frequency point while waiting for the next trigger.

4.6 Power Sweep

lscan_cw_swp: Continuous wave automatic power sweep. Upon receiving a trigger, power sweeping is performed automatically according to the configured step size and dwell time.

lscan_pb_swp: Automatic modulated signal power sweep. Upon receiving a trigger, power levels are switched step-by-step according to the configured step size and dwell time, and the specified baseband waveform is continuously played at each power level.

lscan_pb_hop_1wav: Single-step modulated signal power switching. Power changes step-by-step based on external commands, and the specified waveform is continuously played at the current power level while waiting for the next trigger.

www.harogic.com
info@harogic.com
+65-8299 8857