Arm Virtual Hardware Advanced Features

Version 1.0

Knowledge Base Article

Non-Confidential

Copyright © 2022 Arm Limited (or its affiliates).
All rights reserved.

Issue 01
107688_0100_01_en

Arm Virtual Hardware Advanced Features

Knowledge Base Article

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	21 September 2022	Non-Confidential	First release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws

and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at https://www.arm.com/company/policies/trademarks.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Contents

1. Play Video Files on Virtual i.M X8M Boards.....7
1.1 Table of Contents.....7
1.2 Open the GStreamer Multimedia Application.....7
1.3 Play Video Using the Web Interface.....10
1.4 Use Your Webcam.....10
1.5 Upload a File - Web Interface.....11
1.6 Link to a URL - Web Interface.....12
1.7 Play Videos Using API and POST Requests.....13
1.8 Upload a File - API and POST Requests.....13
1.9 Link to a URL - API and POST Requests.....14

2. Use USB/IP to Attach Peripherals over the Internet.....15
2.1 Table of Contents.....15
2.2 Use Your Local Linux Computer as the USB/IP Host.....15
2.3 Set Up the USB/IP Daemon on Your Computer.....16
2.4 Attach a Physical Device to the Virtual Board Client.....21
2.5 (Optionally) Close the Connection When Finished.....25
2.6 Use a Virtual STM32U5 Board as the USB/IP Host.....28
2.7 Build USB Firmware Files with the STM32CubeIDE Tool.....28
2.8 Attach a Virtual USB Peripheral to a Virtual Raspberry Pi 4 Board.....34
2.9 Attach a Virtual USB Peripheral to a Physical Computer Using a VPN Connection.....43
2.10 Attach a Virtual USB Peripheral to a Physical Computer Using SSH Port Forwarding.....51

3. vMMIO Introduction.....53
3.1 Background.....53
3.2 The CoreIO Library API.....53
3.3 Connecting to a target VM.....53
3.4 Registering a vMMIO Range.....54
3.5 Registering a vMMIO Handler.....54
3.6 Processing Read/Write Events.....55
3.7 Interrupts.....56
3.8 Direct Memory Access (DMA).....57
3.9 Examples.....57

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4. Kernel debugging.....59
4.1 Using GDB/LLDB.....59
4.2 Monitor Commands.....60

5. Custom device tree.....62

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

1. Play Video Files on Virtual i.M X8M Boards

Upload videos to your device or provide a URL to play videos in a loop on your virtual iMX8m board.

1.1 Table of Contents

This section contains the following:

- Open the GStreamer Multimedia Application
- Play Video Using the Web Interface
  - Use Your Webcam
  - Upload a File - Web Interface
  - Link to a URL - Web Interface
- Play Videos Using API and POST Requests
  - Upload a File - API and POST Requests
  - Link to a URL - API and POST Requests

The virtual iMX8m board supports playing `.mp4` video files directly on your device.

You can use either the AVH web interface or a set of API and POST requests.

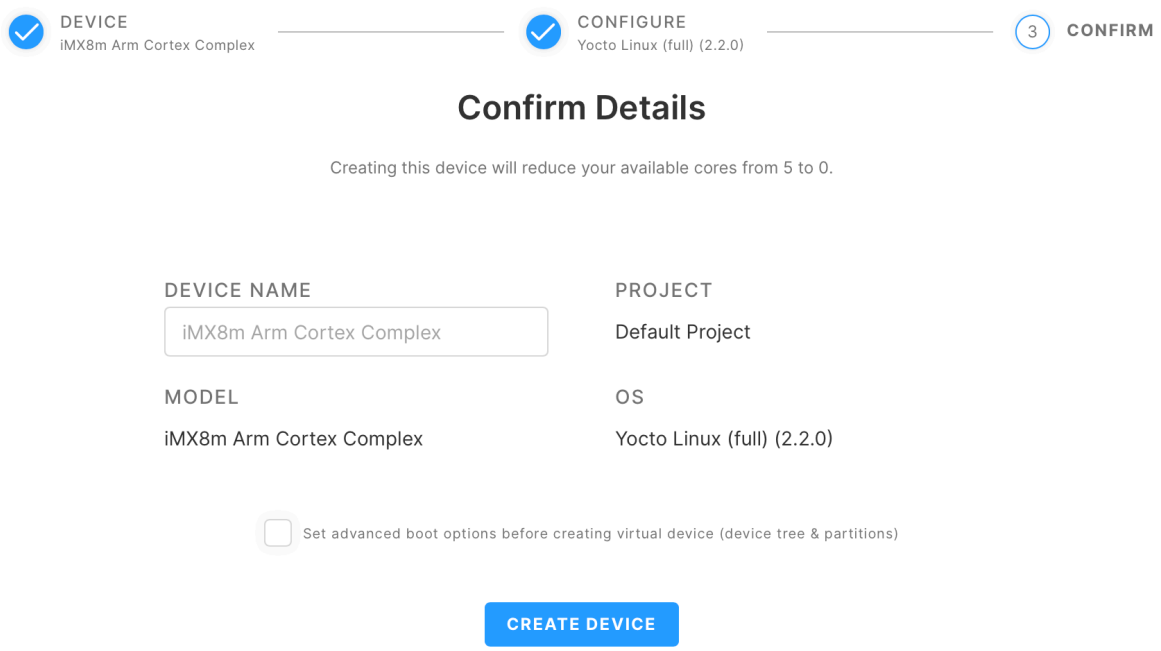
1.2 Open the GStreamer Multimedia Application

To open GStreamer:

1. Create a virtual iMX8M Arm Cortex Complex board running Yocto Linux OS version 2.2.0.

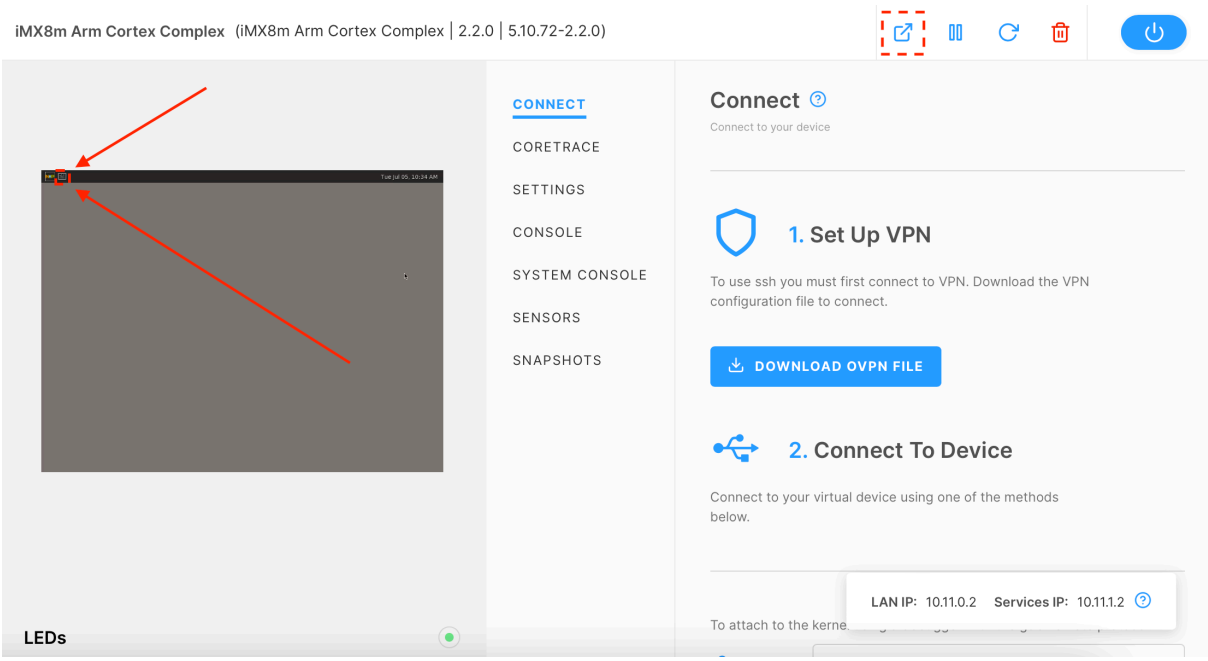
Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 1-1: Create iMX8m Board



2. Open the Terminal by clicking the second icon in the top-left of the device's display. You may prefer to open the display to a new window by clicking Pop Out towards the top-right.

Figure 1-2: Open Terminal

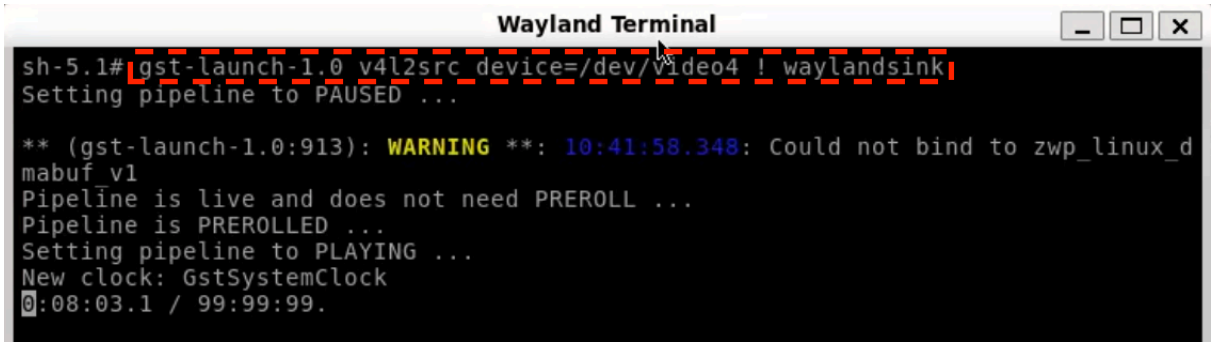


Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3. Run the following command in Terminal to launch GStreamer.

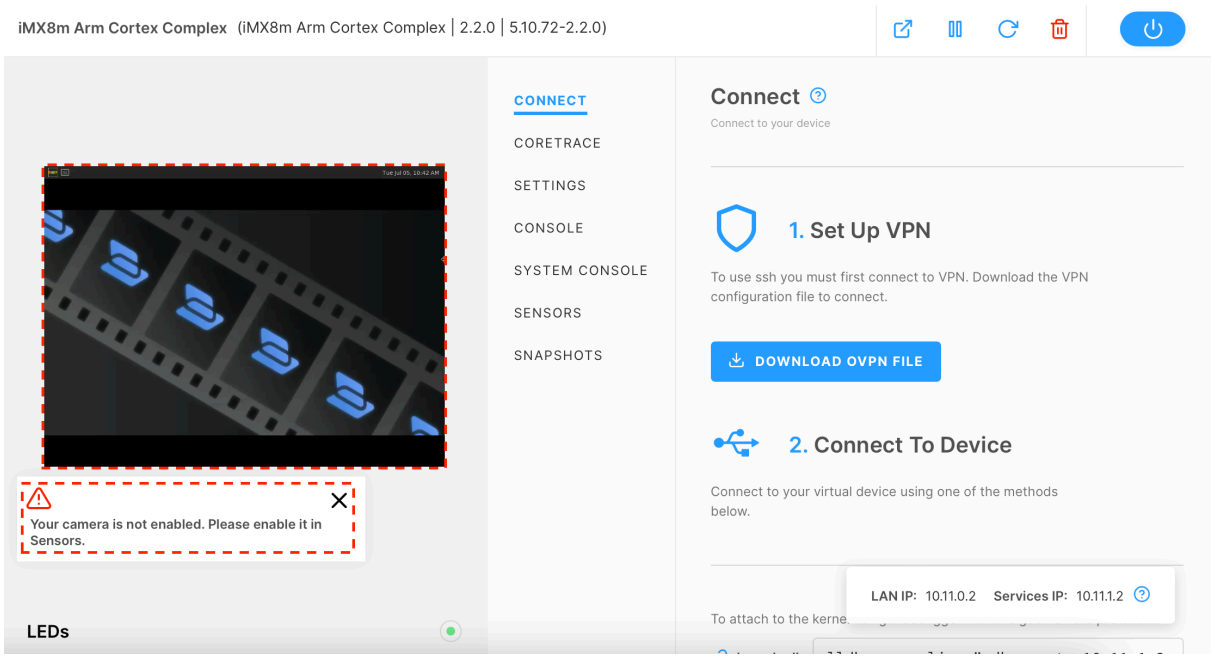
```
gst-launch-1.0 v4l2src device=/dev/video4 ! waylandsink
```

Figure 1-3: Terminal Command



4. After a few seconds, you should see the video player open along with a message that your camera is not enabled.

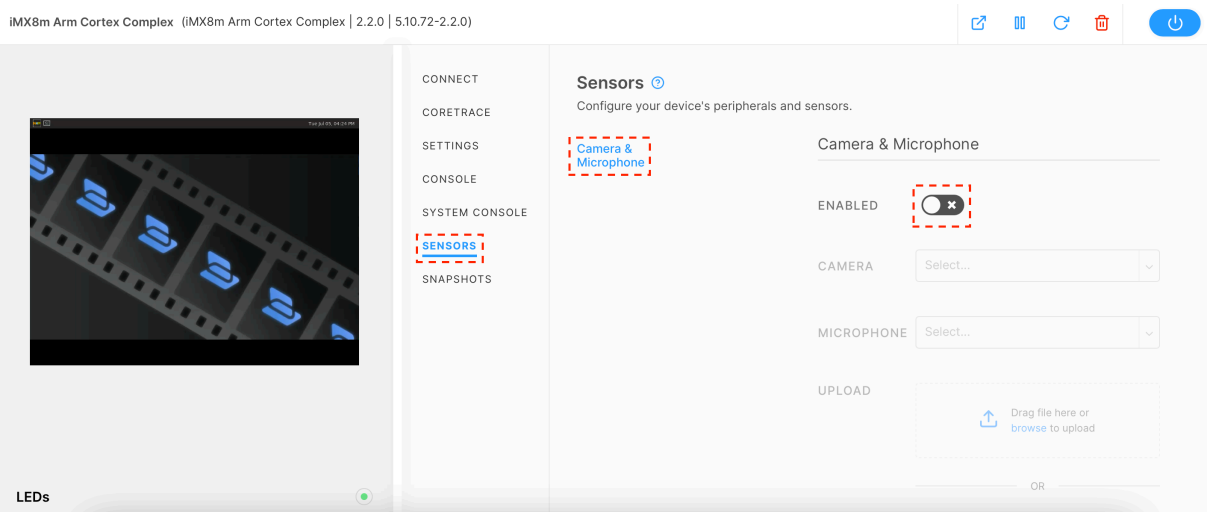
Figure 1-4: GStreamer Opened



5. Open the Camera & Microphone settings under the Sensors tab and enable the sensors.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 1-5: Enable the sensors



1.3 Play Video Using the Web Interface

The iMX8m device supports video playback from your webcam, by uploading a file, or by sharing the video's URL.

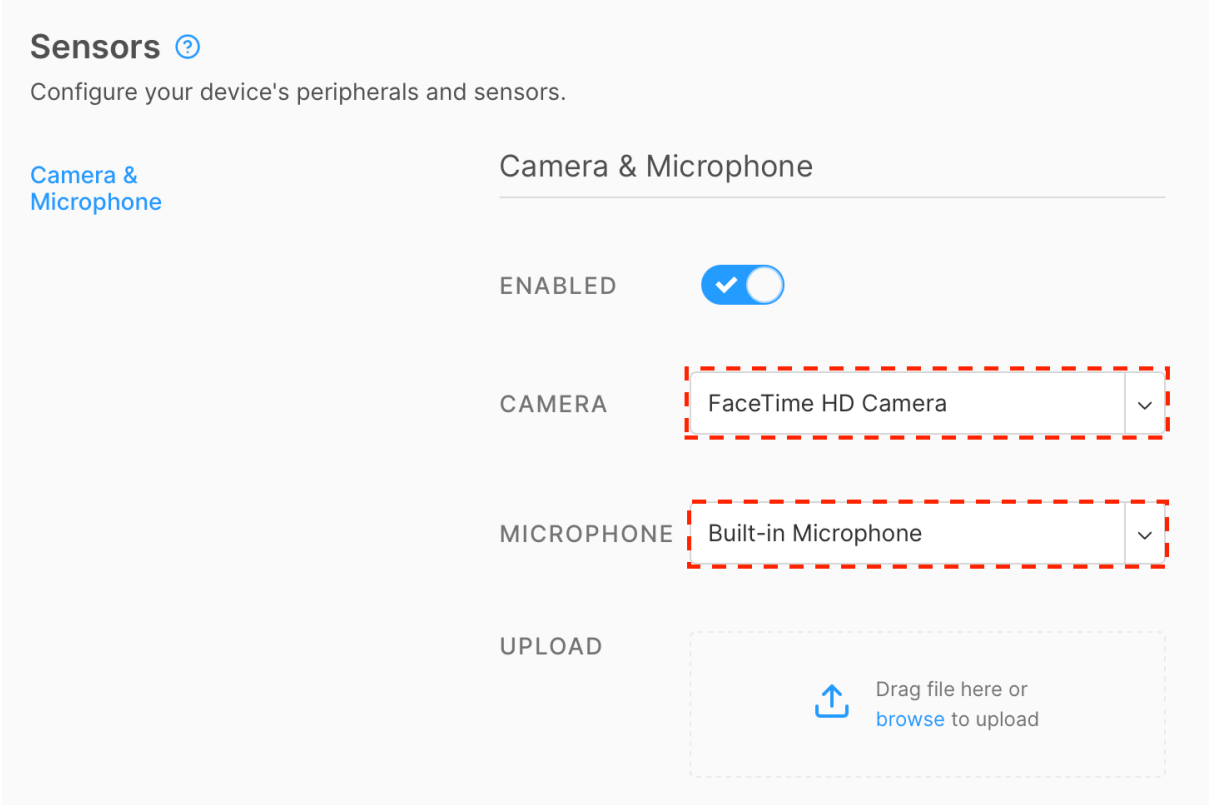
1.4 Use Your Webcam

To use your webcam:

1. After you enable the camera and microphone sensors, allow the browser to access your computer's built-in webcam and microphone. (This process will vary depending on your browser.)
2. Confirm that you can see your computer's sensors appear under CAMERA and MICROPHONE.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 1-6: Confirm sensors



3. You should see your webcam video appear on GStreamer.

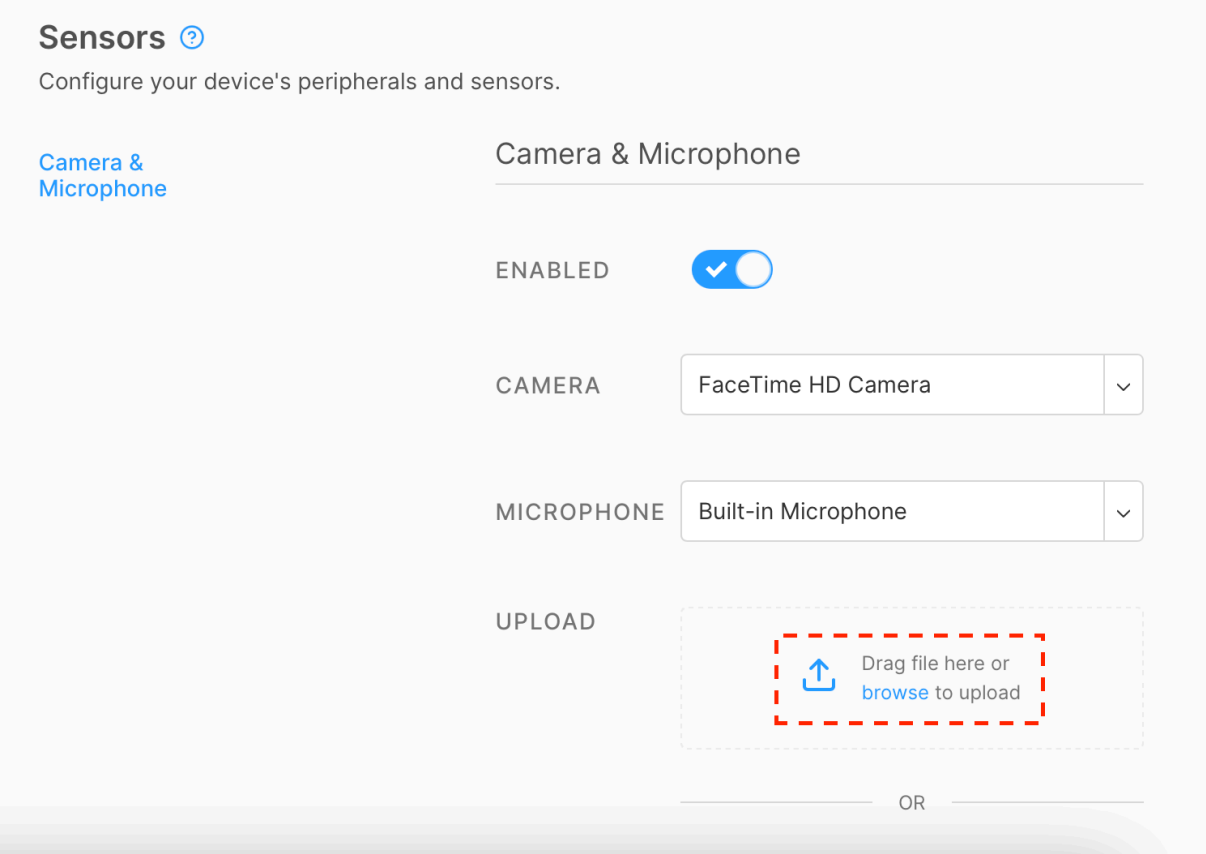
1.5 Upload a File - Web Interface

To upload a file using the web interface:

1. After enabling the virtual camera and microphone in the Sensors tab, upload the `.mp4` video file. You do not need to allow the browser to access your local computer's physical camera and microphone.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 1-7: Upload the `.mp4` video file



2. When the upload completes, click PLAY VIDEO. The video should play on GStreamer in a loop.

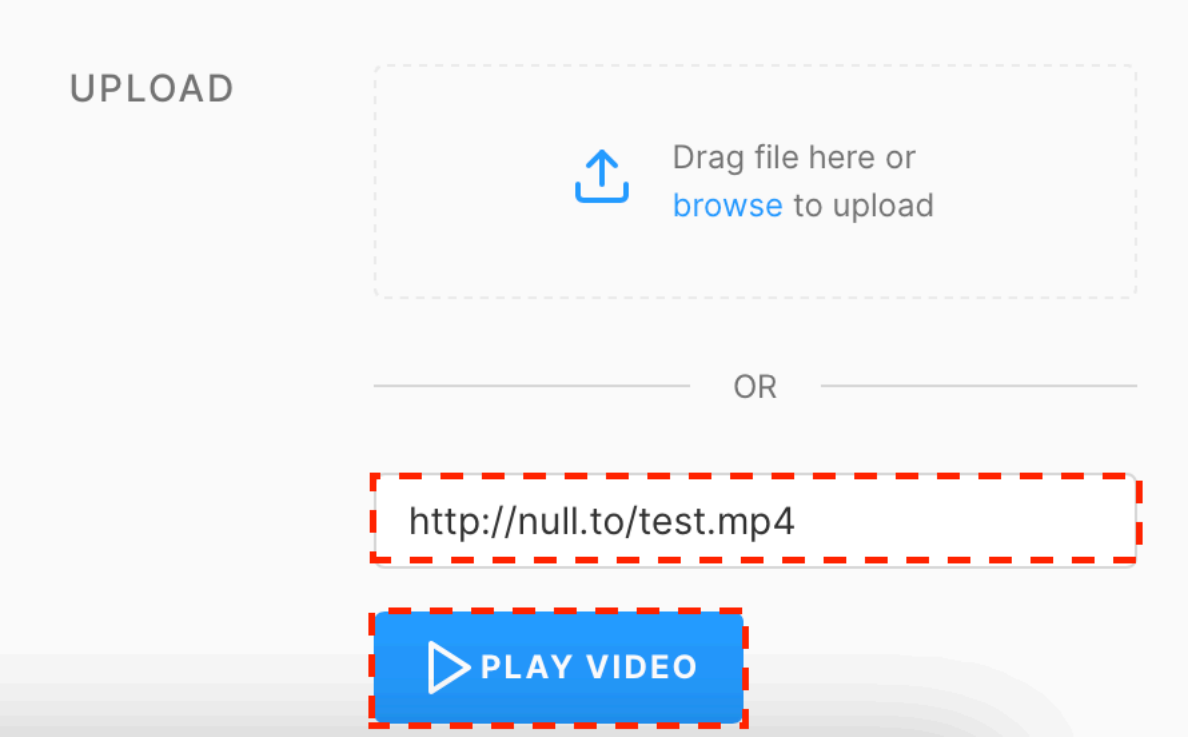
1.6 Link to a URL - Web Interface

To link to a URL using the web interface:

1. After enabling the camera and microphone in the Sensors tab, enter a URL to a `.mp4` video and click PLAY VIDEO. You do not need to allow the browser to access your local computer's physical camera and microphone.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 1-8: Enter a URL



2. The video should play on GStreamer in a loop.

1.7 Play Videos Using API and POST Requests

You can also use the command line to play multimedia files.

Start with GStreamer opened (`gst-launch-1.0 v4l2src device=/dev/video4 ! waylandsink`).

1.8 Upload a File - API and POST Requests

To upload a file using the AVH API:

1. Use the AVH API to acquire the device instance.
2. Upload the `.mp4` file to that instance.

```
const videoImage = await instance.uploadImage('media',
 $LOCATION_OF_AN_MP4_ON_YOUR_LOCAL_DISK, '$ANY_ARBITRARY_STRING')
```

3. Note the `imageId` that is returned.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4. Make a POST request to `/api/v1/instances/${instanceId}/play` with the following body.

```
{
  instanceId,
  imageId
}
```

5. Verify that you can see the video playing.

1.9 Link to a URL - API and POST Requests

To link to a URL using the AVH API:

1. Use the AVH API to acquire the device instance.
2. Make a POST request to `/api/v1/instances/${instanceId}/play` with the video URL in the body.

```
{
  instanceId,
  url: 'http://null.to/test.mp4'
}
```

3. Verify that you can see the video playing.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2. Use USB/IP to Attach Peripherals over the Internet

The USB/IP protocol allows you to share USB devices between your virtual machine and another device.

2.1 Table of Contents

This section contains the following:

- Use Your Local Linux Computer as the USB/IP Host
  - Set Up the USB/IP Daemon on Your Computer
  - Attach a Physical Device to the Virtual Board Client
  - (Optionally) Close the Connection When Finished
- Use a Virtual STM32U5 Board as the USB/IP Host
  - Build USB Firmware Files with the STM32CubeIDE Tool
  - Attach a Virtual USB Peripheral to a Virtual Raspberry Pi 4 Board
  - Attach a Virtual USB Peripheral to a Physical Computer Using a VPN Connection
  - Attach a Virtual USB Peripheral to a Physical Computer Using SSH Port Forwarding

USB/IP is an open-source tool for encapsulating USB I/O messages into TCP/IP messages. The host device shares a USB peripheral that the client device can use as if it were directly attached to the client. Read more about the protocol on the USB/IP SourceForge page.

In this article, we will show an example of how to Use Your Local Linux Computer as the USB/IP Host to directly control the cursor on a virtual Raspberry Pi Desktop device over a VPN connection.

We will also show examples of how to Use a Virtual STM32U5 Board as the USB/IP Host to share both input and output USB peripherals.

2.2 Use Your Local Linux Computer as the USB/IP Host

We will be using a physical computer with a fresh install of Ubuntu Desktop 22.04 LTS and a Raspberry Pi 4 running the Desktop version of firmware 11.2.0.

Start with the USB peripheral disconnected from your device.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

The USB device we use will stop working on your host, so make sure to have a second mouse (or laptop trackpad) to control your computer.

2.3 Set Up the USB/IP Daemon on Your Computer

Start with a fresh install of the most recent LTS version of Ubuntu Desktop as the host and a virtual Raspberry Pi 4 Desktop board as the client.

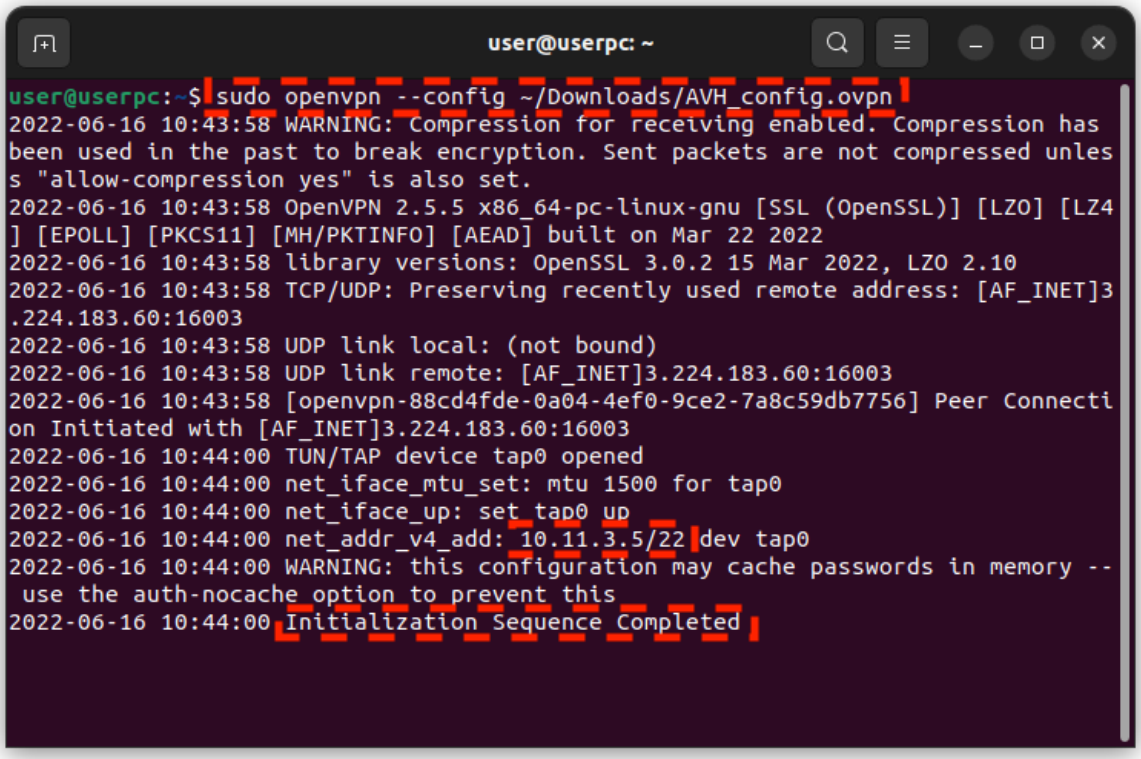
1. Install the required packages.

```
sudo apt update && sudo apt install -y linux-tools-$(uname -r) hwdata ubuntu-restricted-extras
```

2. Download the `.ovpn` file from the Raspberry Pi 4's Connect tab and connect to your AVH project using openvpn. Make note of the VPN's IP address, which in our example is `10.11.3.5`.

```
sudo openvpn --config ~/Downloads/AVH_config.ovpn
```

Figure 2-1: Connect to AVH VPN

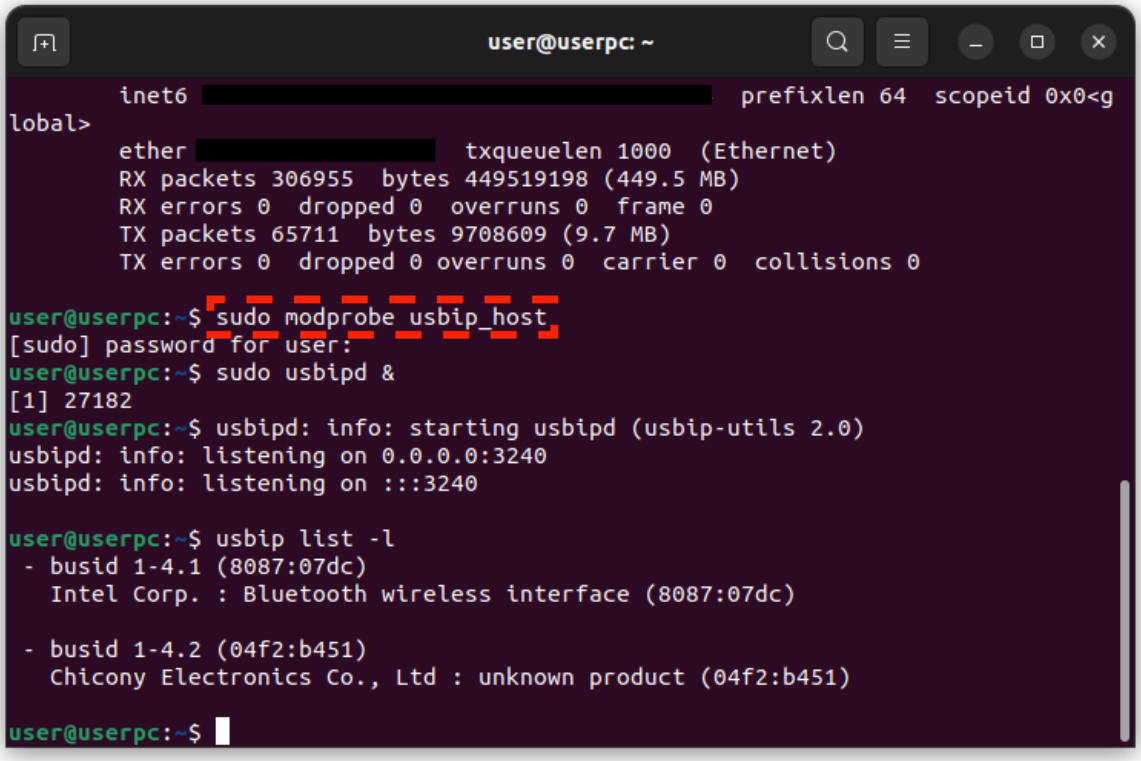


Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3. Open a new Terminal window and load the USB/IP Host kernel module.

```
sudo modprobe usbip_host
```

Figure 2-2: Load the USBIP host kernel package

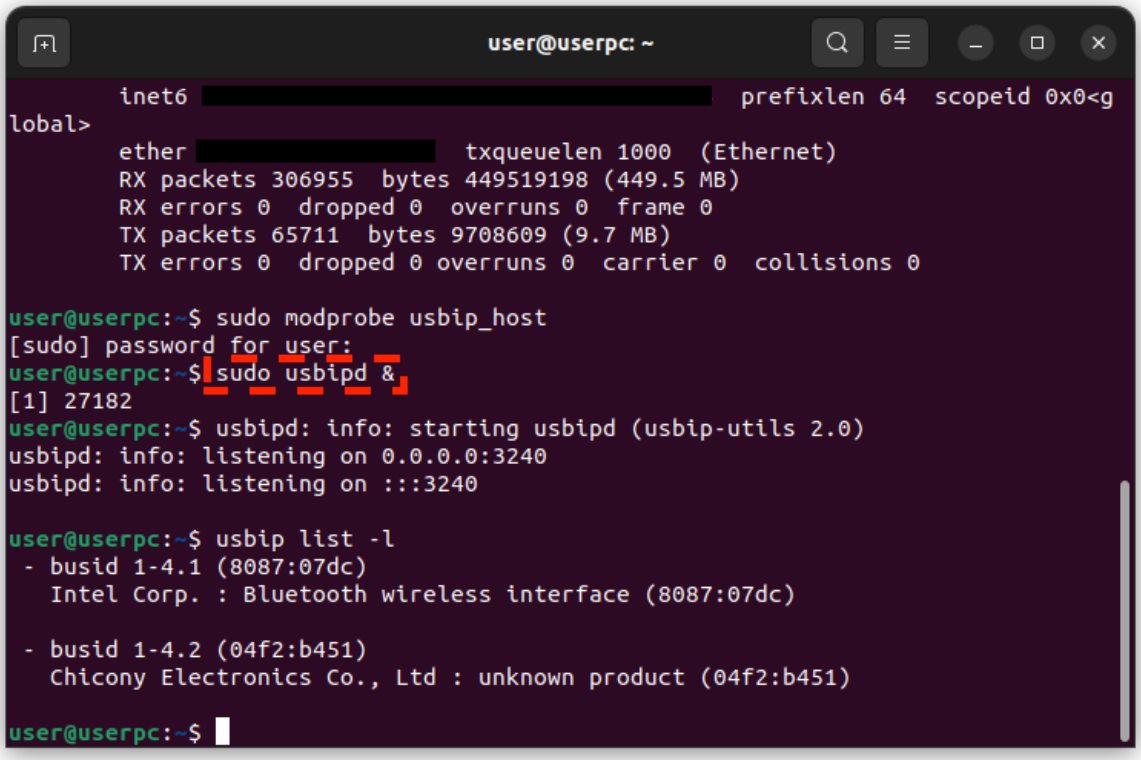


4. Start the USB/IP daemon in the background.

```
sudo usbipd &
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-3: Run the USBIP daemon

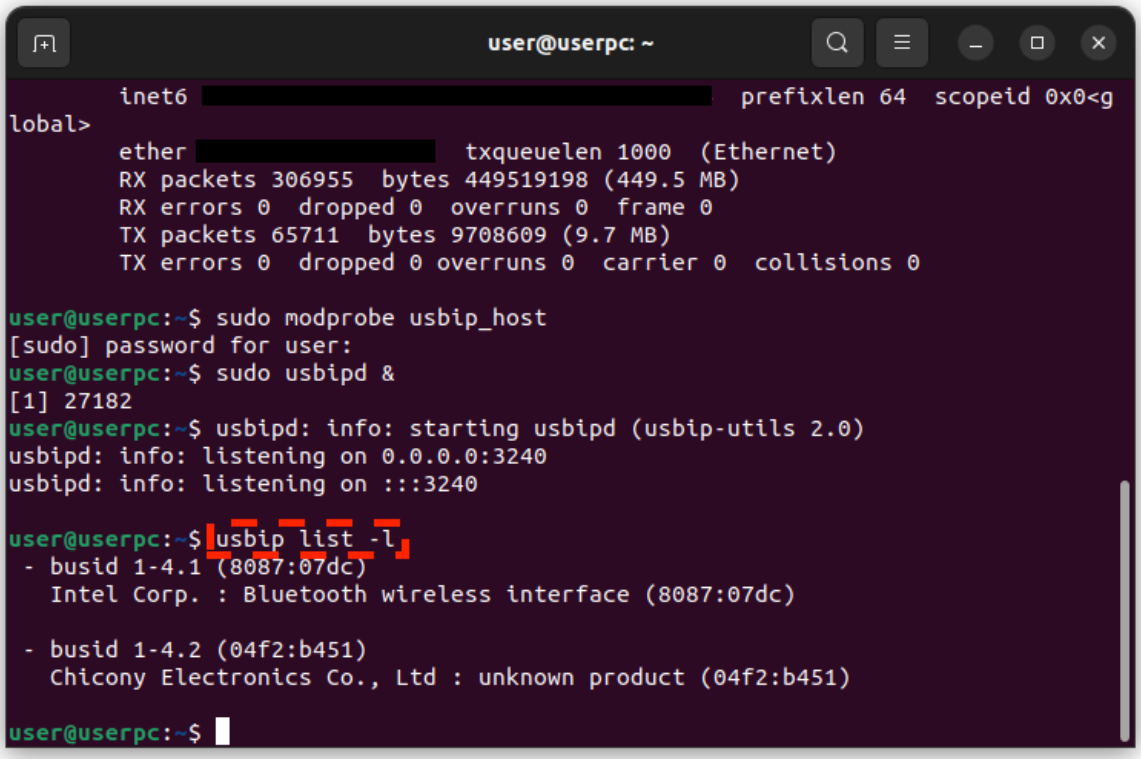


5. With the USB mouse dongle still disconnected, list the available peripherals.

```
usbip list -l
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

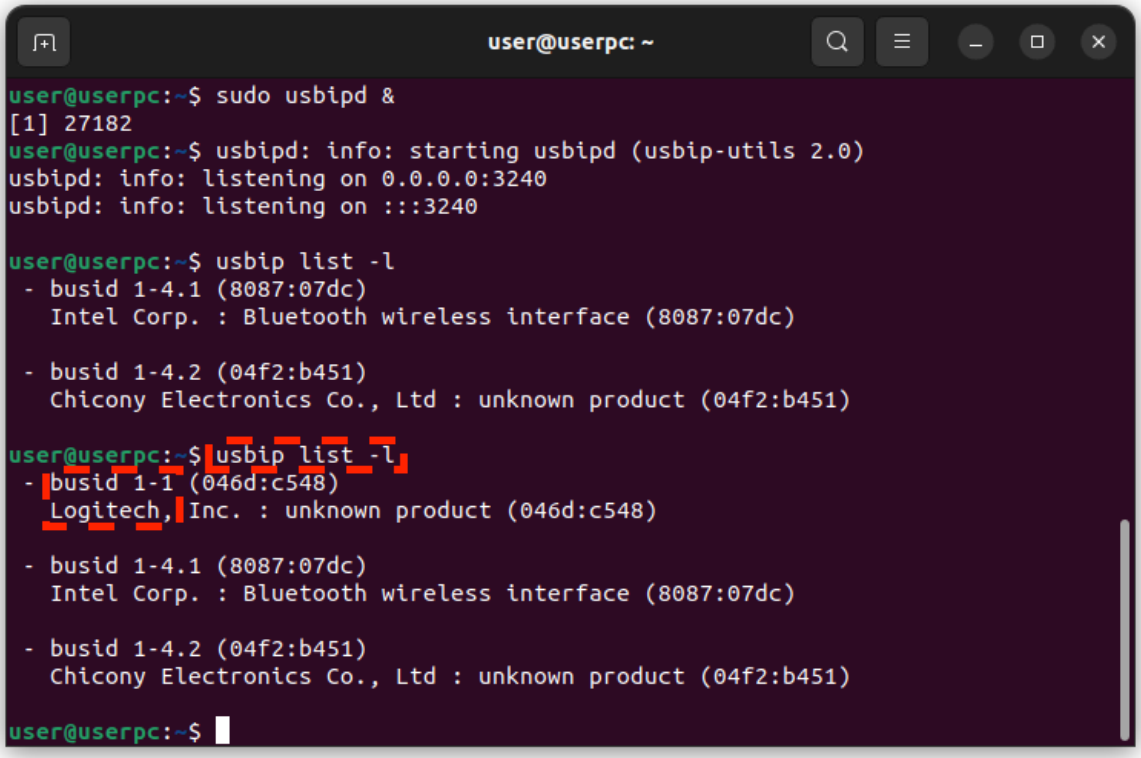
Figure 2-4: USBIP list without device



6. Connect the physical USB mouse dongle to your computer and run the `list` command again. You should see a new device on a new bus number. In our example, we have a new device on bus `1-1`.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-5: USBIP list with device

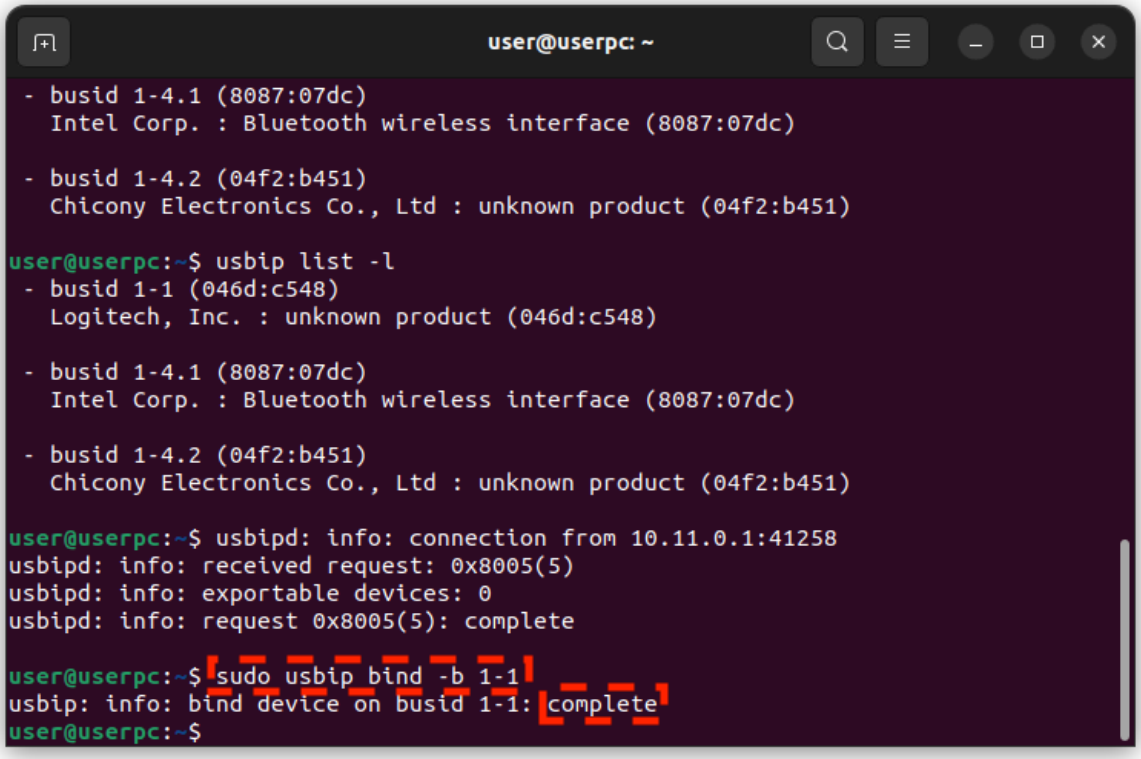


7. Bind the appropriate USB bus, `1-1` in our case. (This will cause the device to stop functioning on your host computer, so make sure to have a backup in the case of a mouse.)

```
sudo usbip bind -b 1-1
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-6: USBIP bind



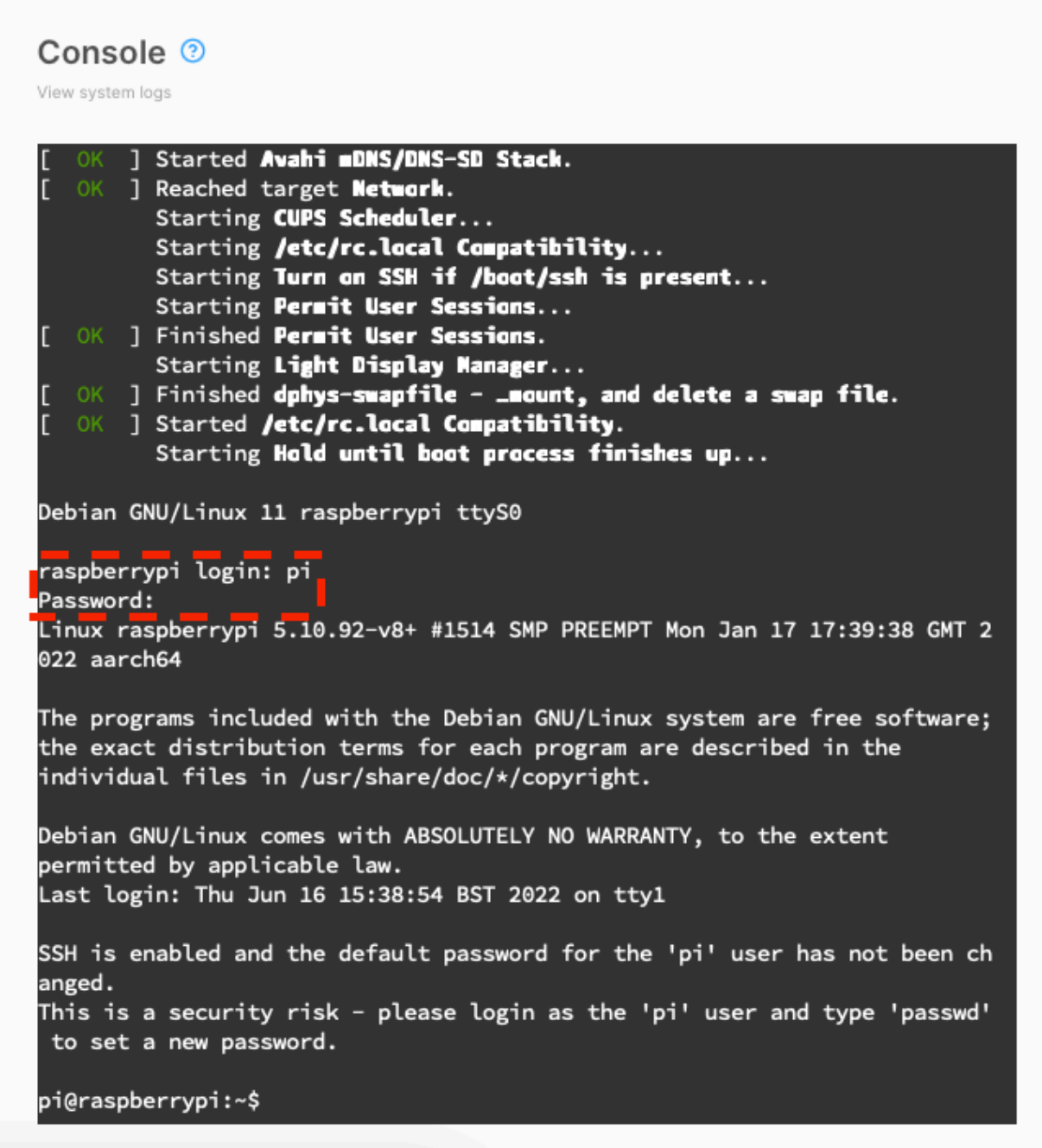
2.4 Attach a Physical Device to the Virtual Board Client

To attach a physical device to the virtual board client:

1. On your newly created Raspberry Pi 4 virtual board, log in with the credentials `pi` and `raspberry`.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-7: Log in to RPi 4 Desktop



2. Install the USB/IP package.

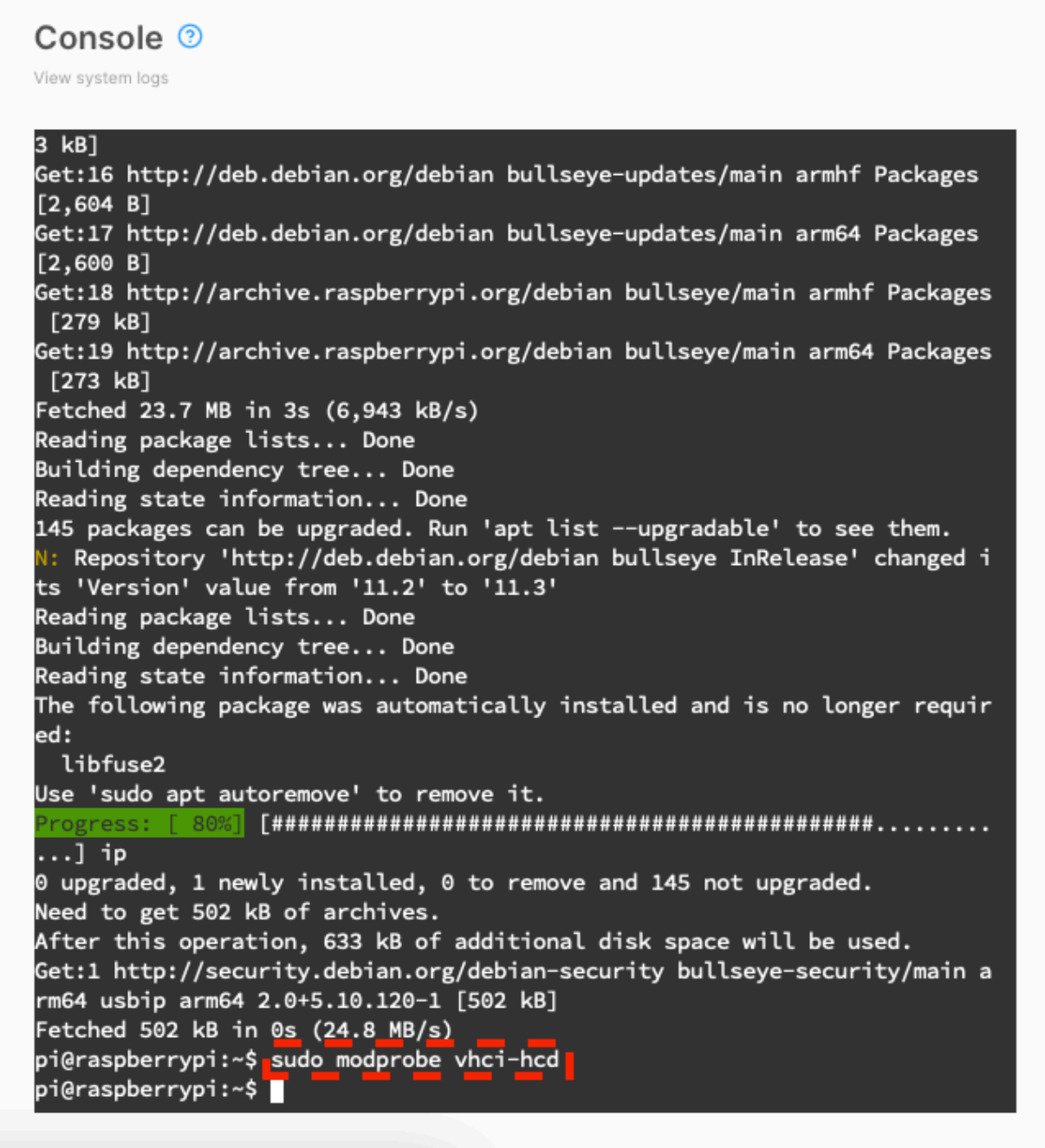
```
sudo apt update && sudo install usbip
```

3. Load the Virtual Host Controller Interface kernel module.

```
sudo modprobe vhci-hcd
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-8: Load the VHCI kernel module

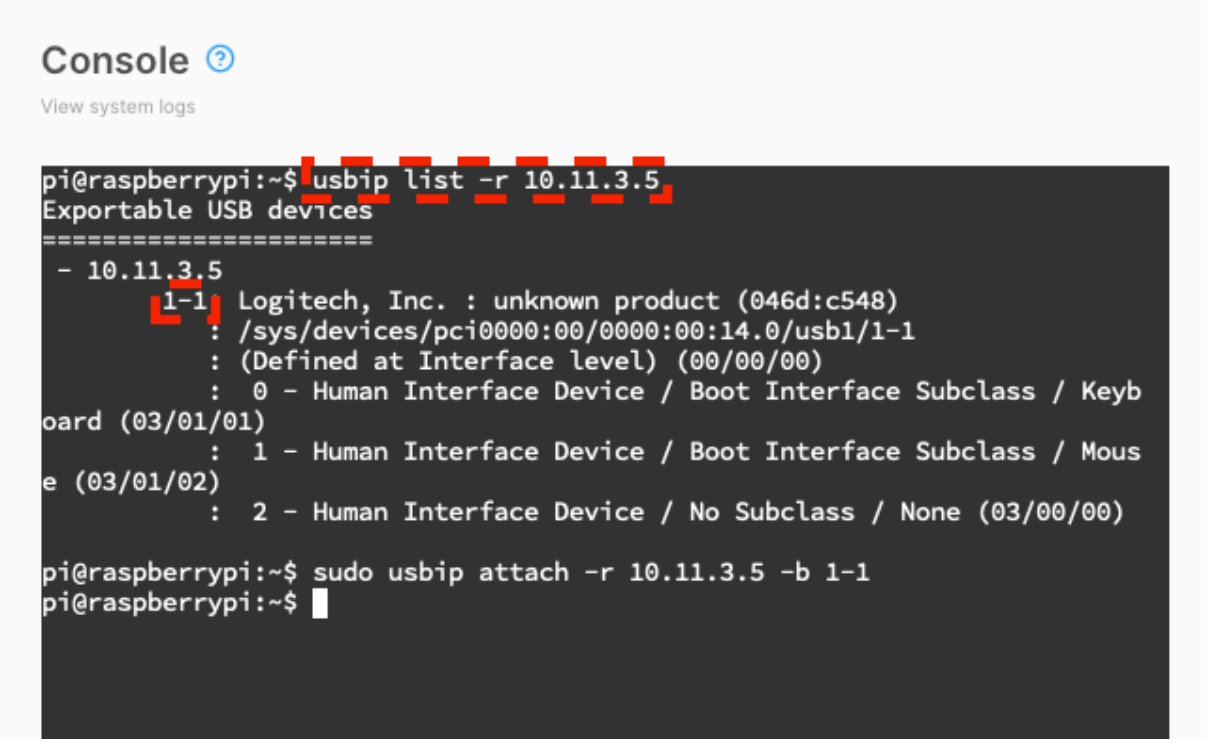


4. Using the VPN IP address from openvpn, list the exportable USB devices on the host. Look for the same bus number from the previous section.

```
usbip list -r 10.11.3.5
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-9: USBIP list on RPi4 Desktop

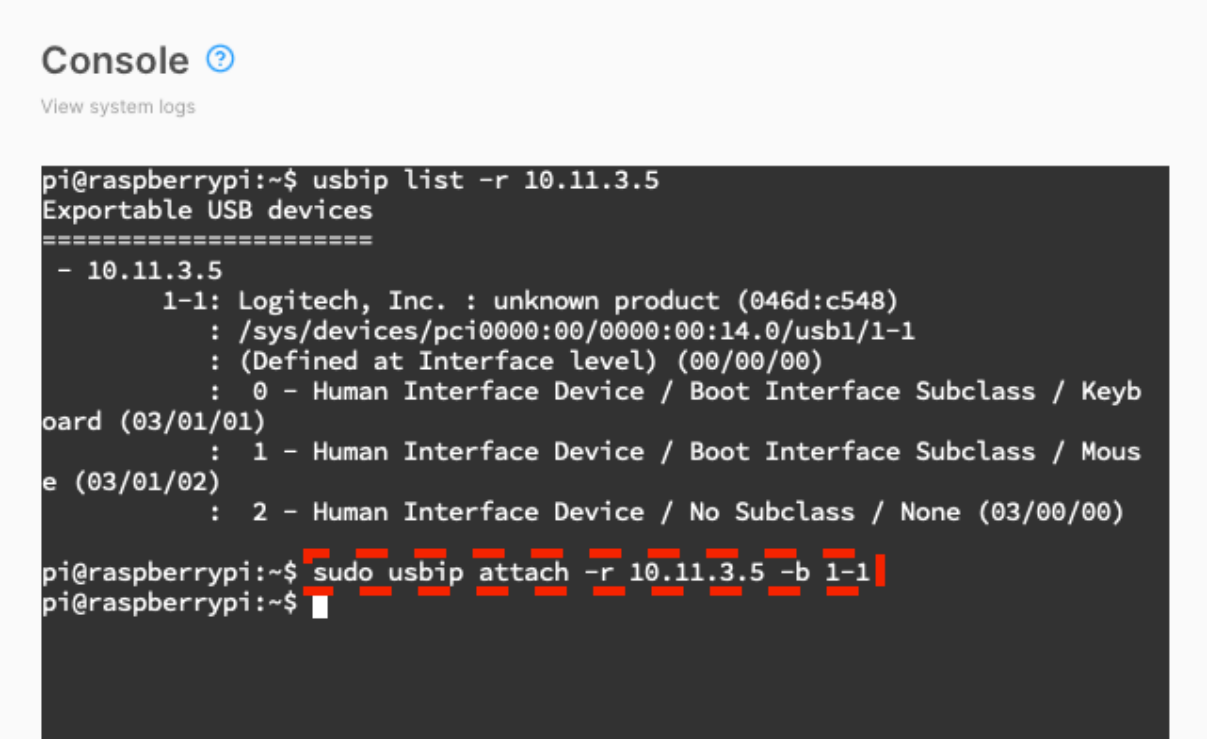


5. Attach to USB bus `1-1`, which corresponds to our mouse.

```
sudo usbip attach -r 10.11.3.5 -b 1-1
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-10: USBIP attach on RPi4 Desktop



6. You should now be able to control the cursor on the Raspberry Pi 4 Desktop.

2.5 (Optionally) Close the Connection When Finished

When you are finished, you can optionally close the connection by detaching on the client and unbinding the host's USB bus.

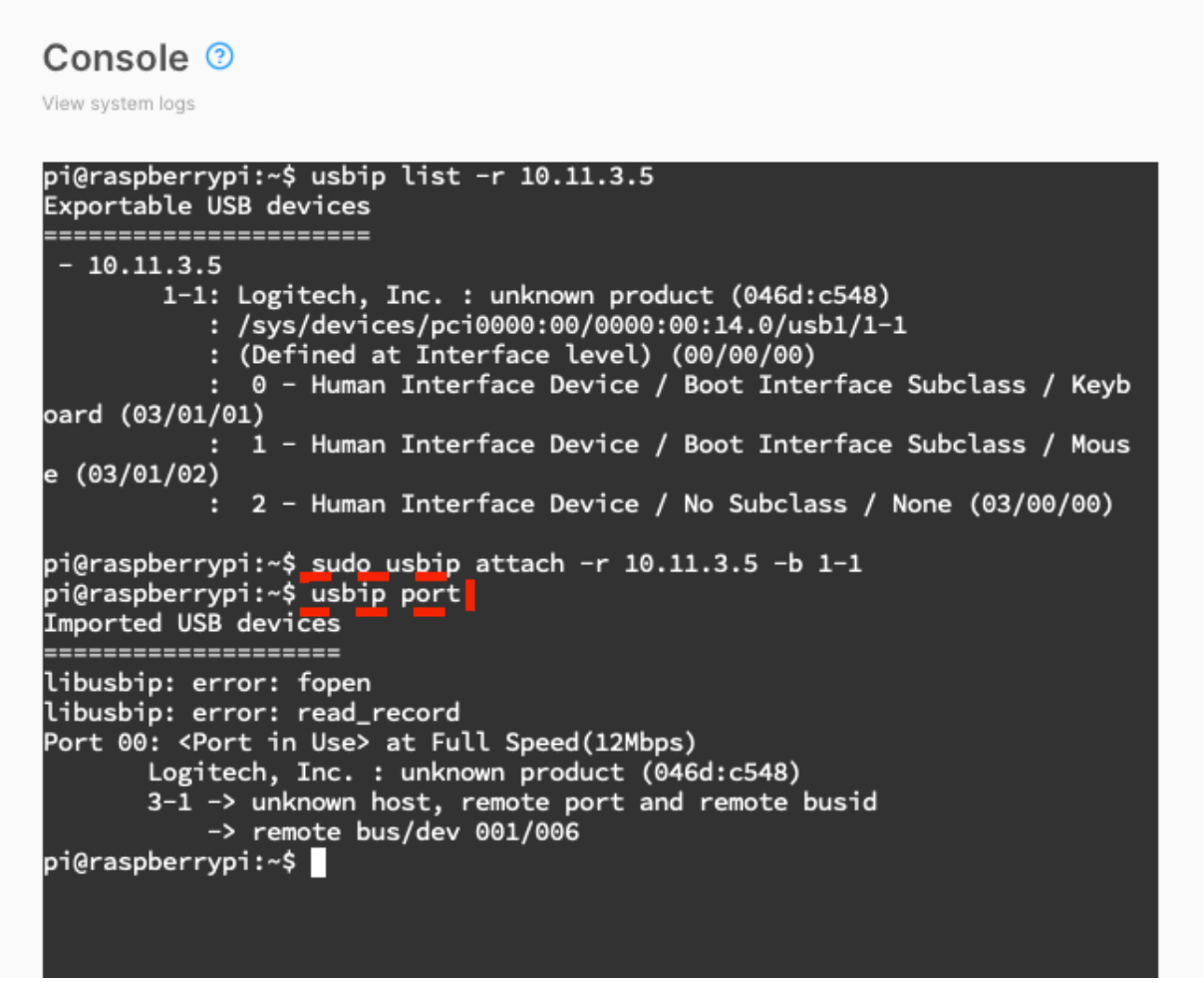
This process is necessary if you want to access the USB resource again on your host device.

1. On the USB/IP client, which in our case is the virtual Raspberry Pi 4 Desktop board, list the imported USB devices. Look for the appropriate port number.

```
usbip port
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-11: USBIP port on RPi4 Desktop

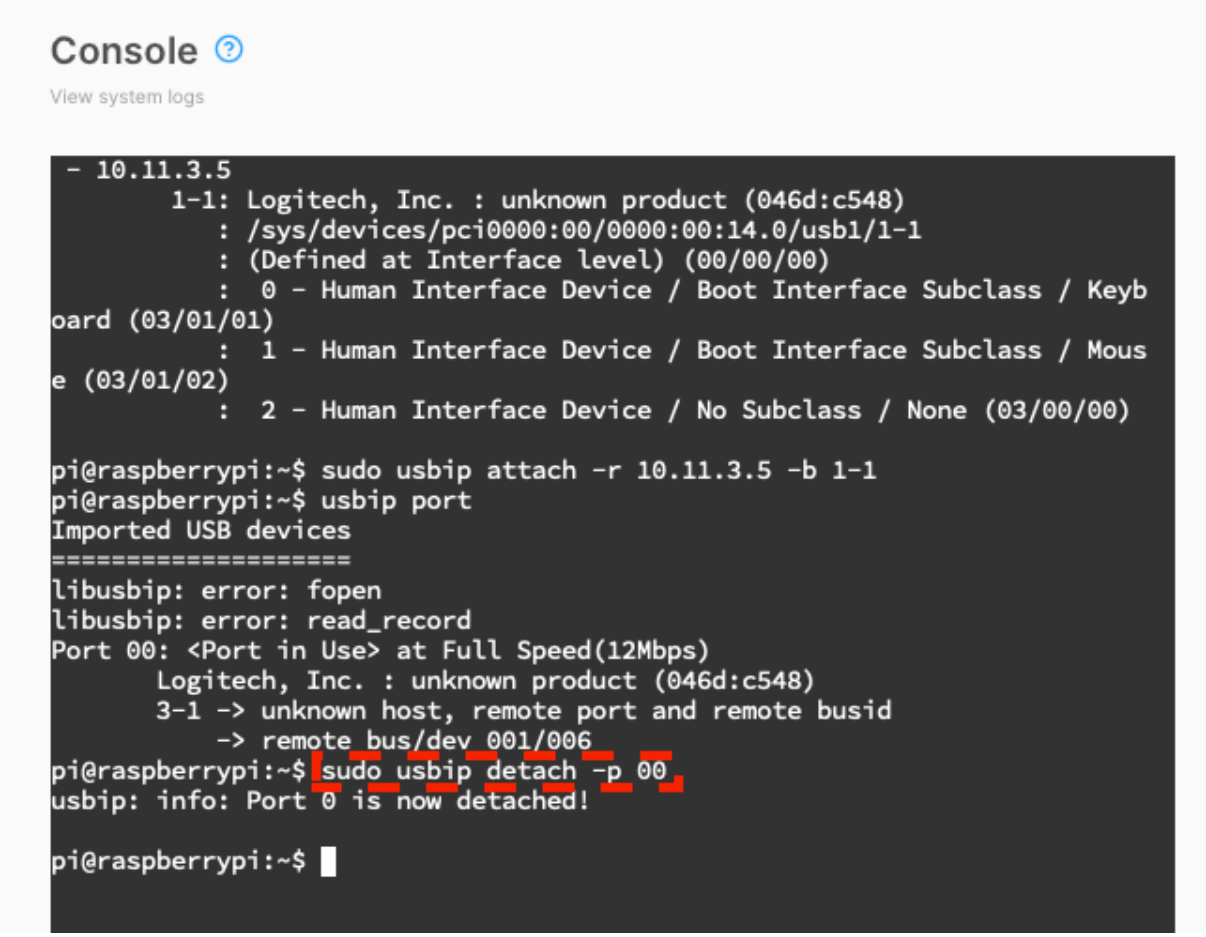


2. Detach port `00`, which corresponds to our mouse. Look for a response showing `detached`.

```
sudo usbip detach -p 00
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-12: USBIP detach on RPi4 Desktop

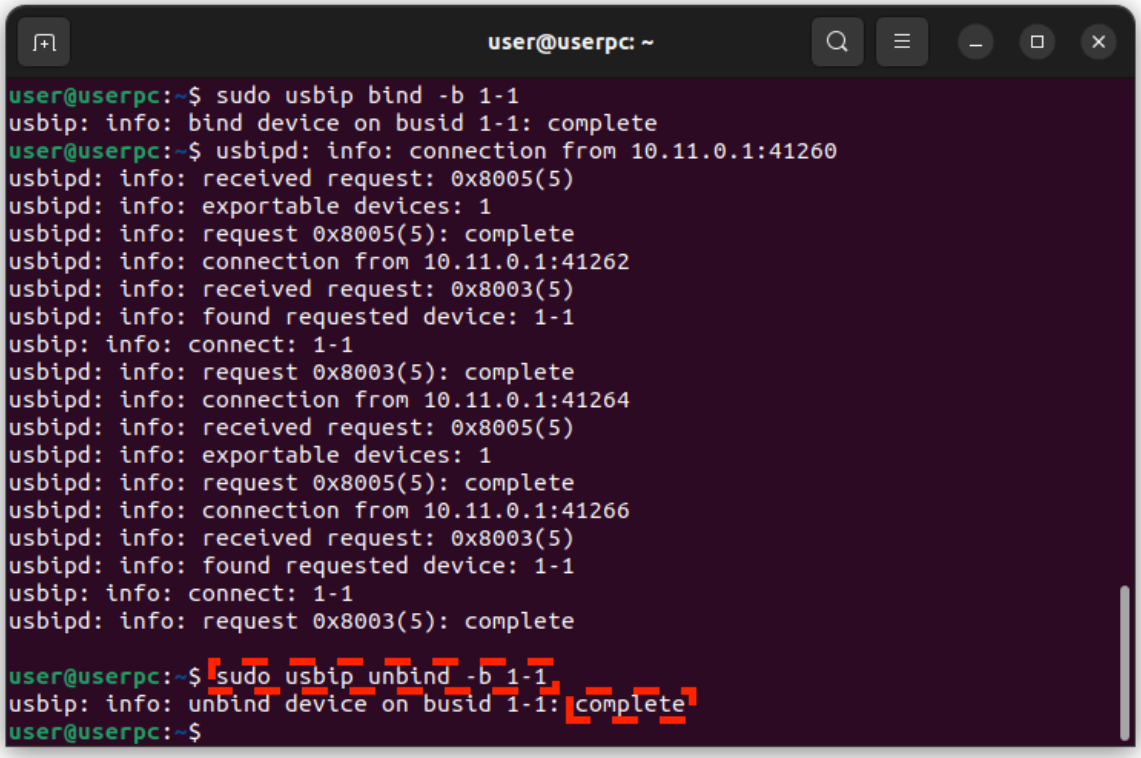


3. Switch over to your USB/IP host, which in our case is the Ubuntu machine, and unbind the appropriate USB bus. Look for a response showing `complete`.

```
usbip unbind -b 1-1
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-13: USBIP unbind



2.6 Use a Virtual STM32U5 Board as the USB/IP Host

We will be using firmware files that are offered as examples in STMicroelectronics's STM32CubeIDE tool.

You will need to build the `Ux_Device_HID.elf` and `Ux_Device_HID_CDC_ACM.elf` firmware files.

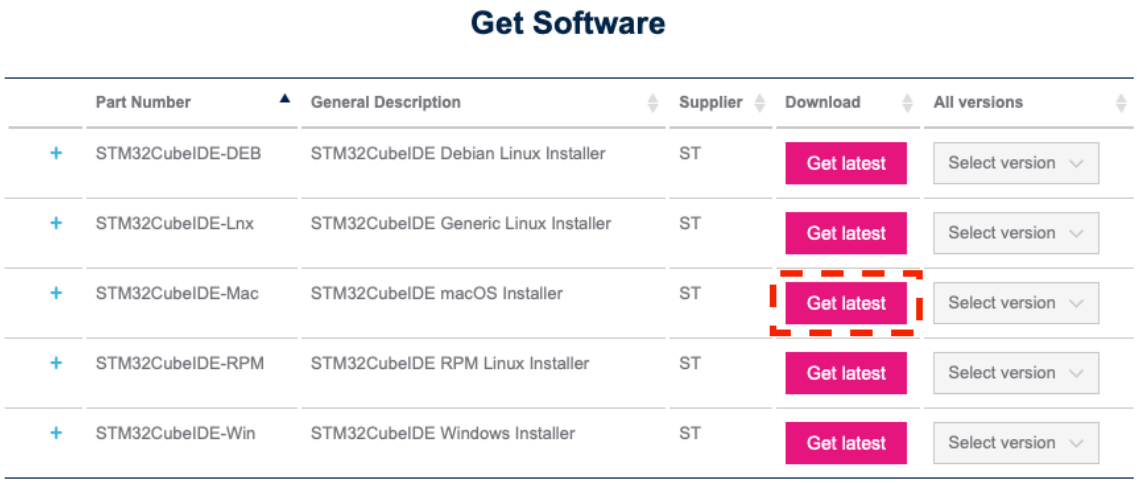
2.7 Build USB Firmware Files with the STM32CubeIDE Tool

Follow this process to build the required `.elf` custom firmware files using free tools from STMicroelectronics.

1. Install the STM32CubeIDE tool. In our example, we will use the MacOS version.

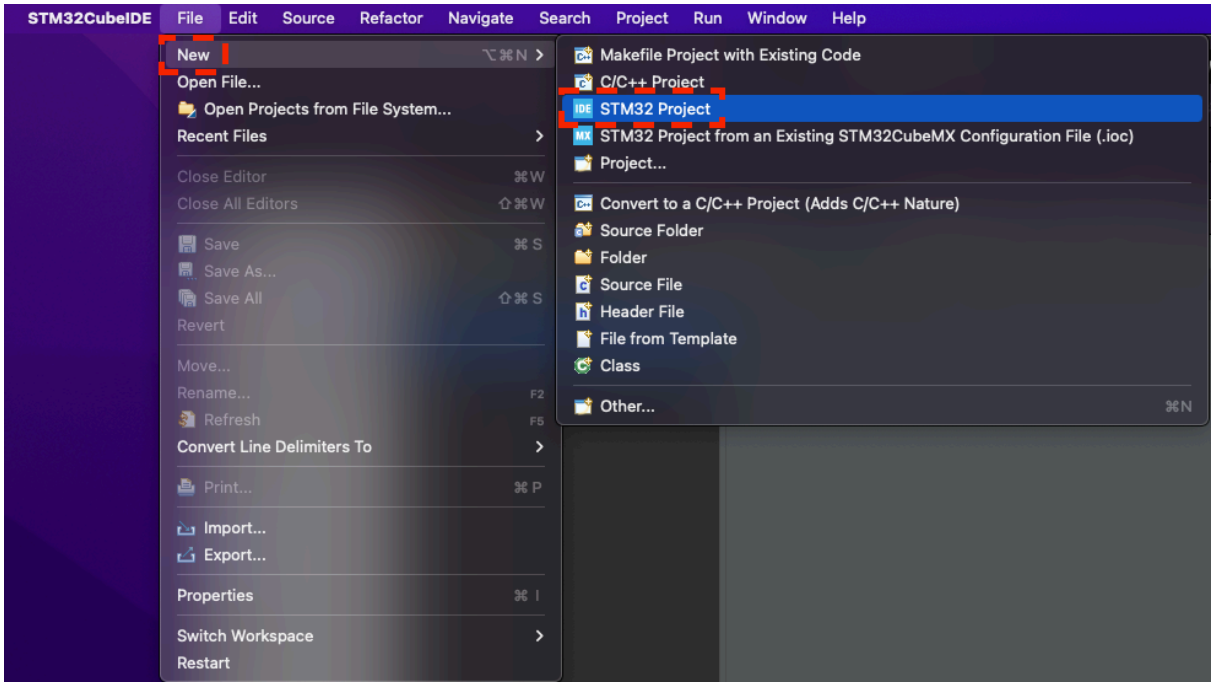
Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-14: Download CubeIDE



2. Create a new project by clicking New -> STM32 Project from the File menu.

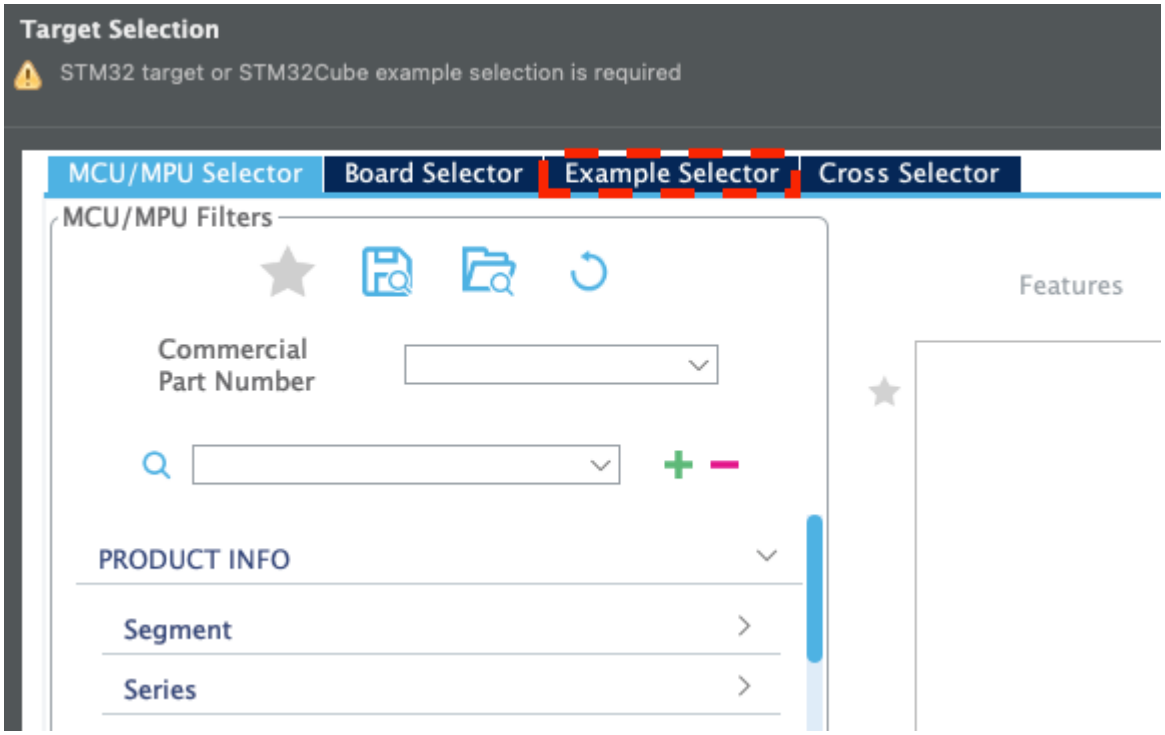
Figure 2-15: New Project



3. When the STM32 Project window appears, click on Example Selector.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

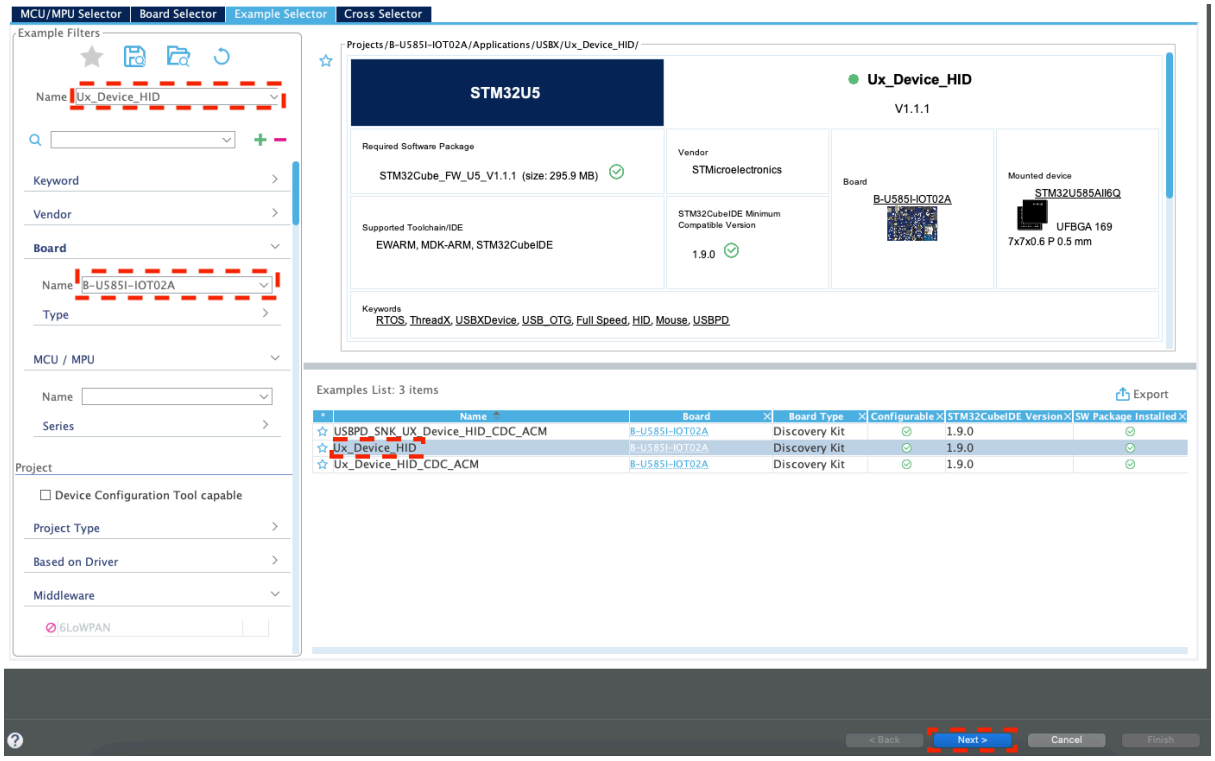
Figure 2-16: Example Selector



4. Choose the example name `Ux_Device_HID` and the board name `B-U585I-IOT02A`, click on the example you want to build, then click Next.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

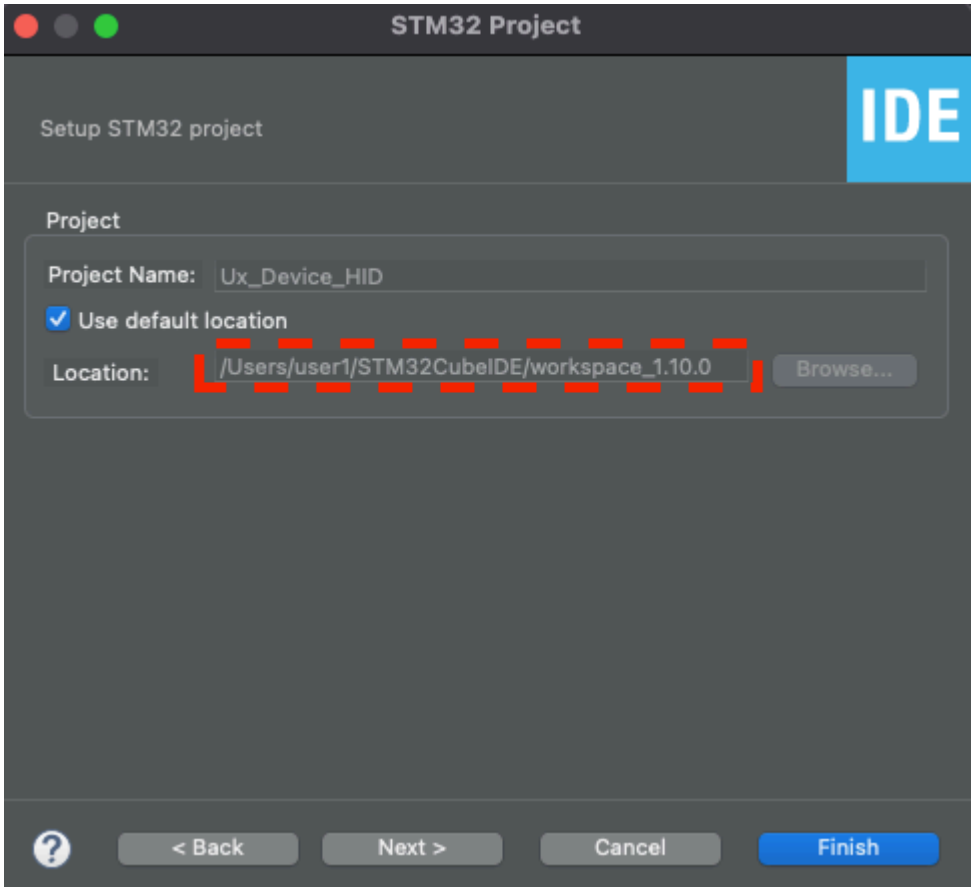
Figure 2-17: Choose Example



5. Make note of your project name and path to your workspace then click Finish.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

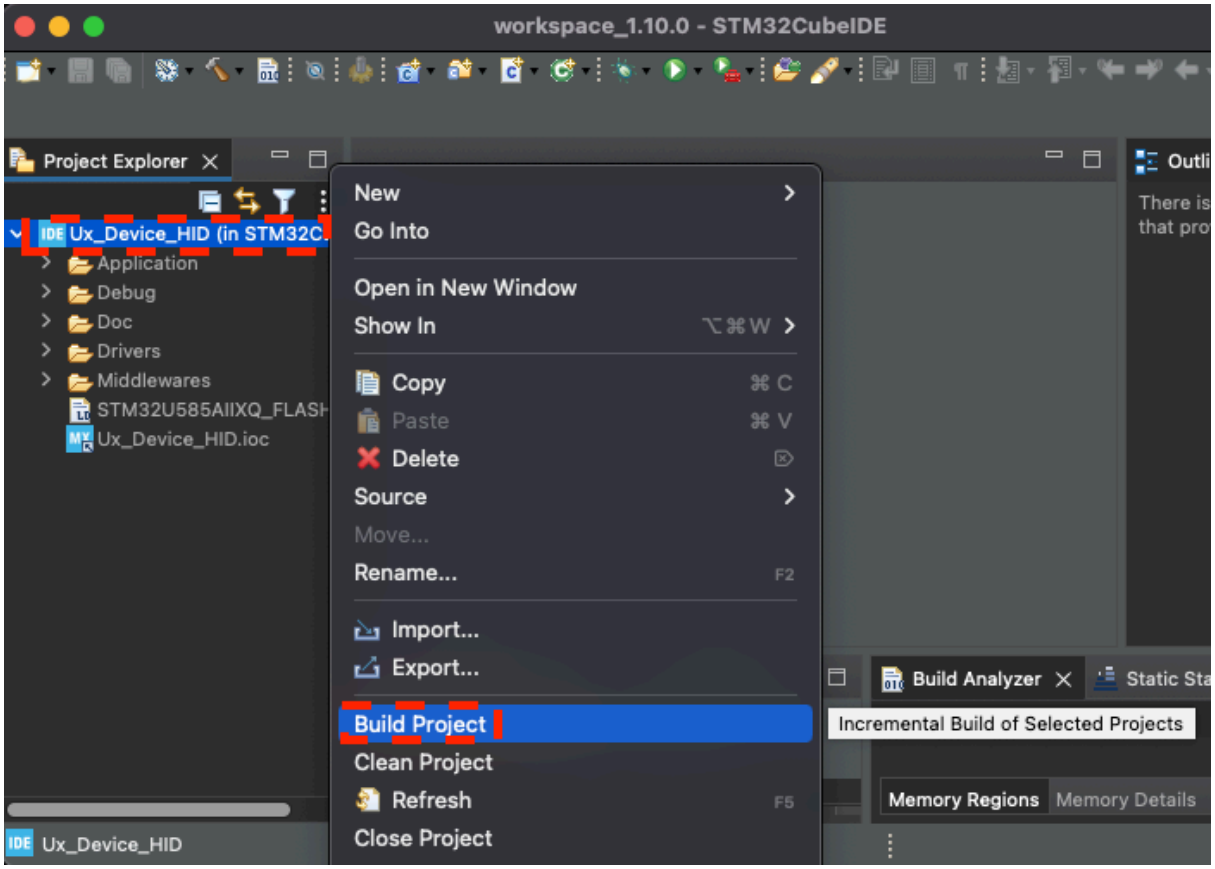
Figure 2-18: Finish Project



6. When the Project Explorer opens, right-click on your project name and select Build Project.

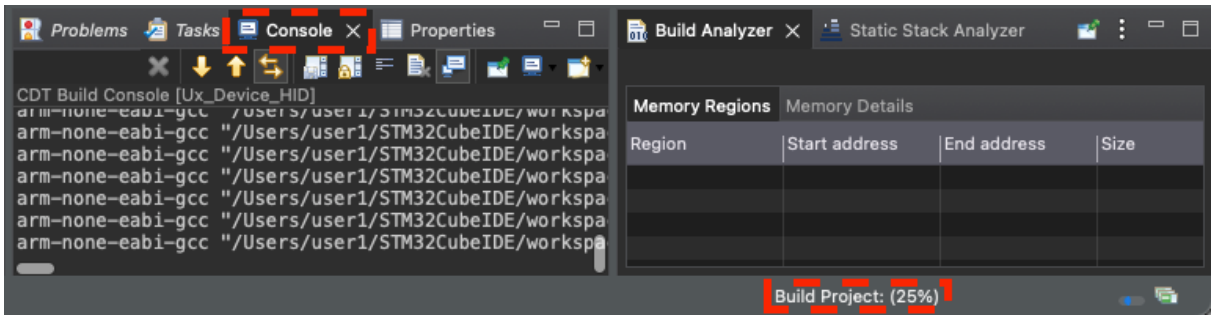
Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-19: Build Project



7. While the project is building, you should see a progress message towards the bottom right. Open the Console if you prefer to follow the progress.

Figure 2-20: Build Progress

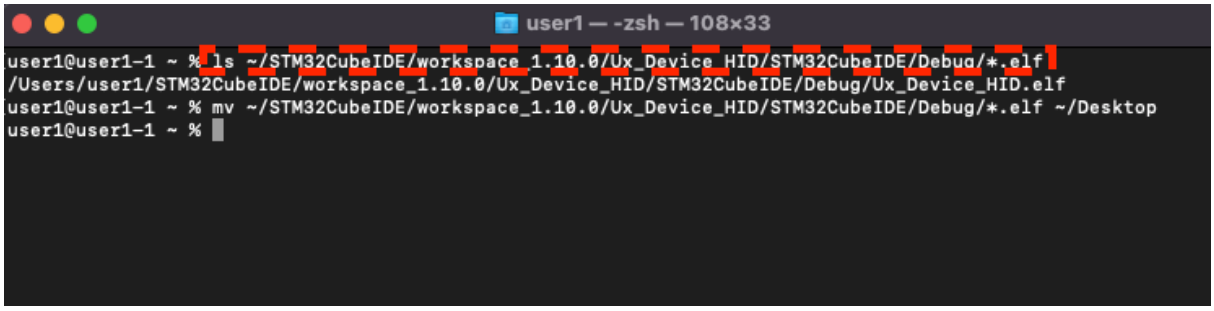


8. The `.elf` firmware file will be created inside your project directory, which is inside `/<path_to_workspace>/<project_name>/STM32CubeIDE/Debug`.

```
ls ~/STM32CubeIDE/workspace_1.10.0/Ux_Device_DFU/STM32CubeIDE/Debug/*.elf
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-21: List Firmware File



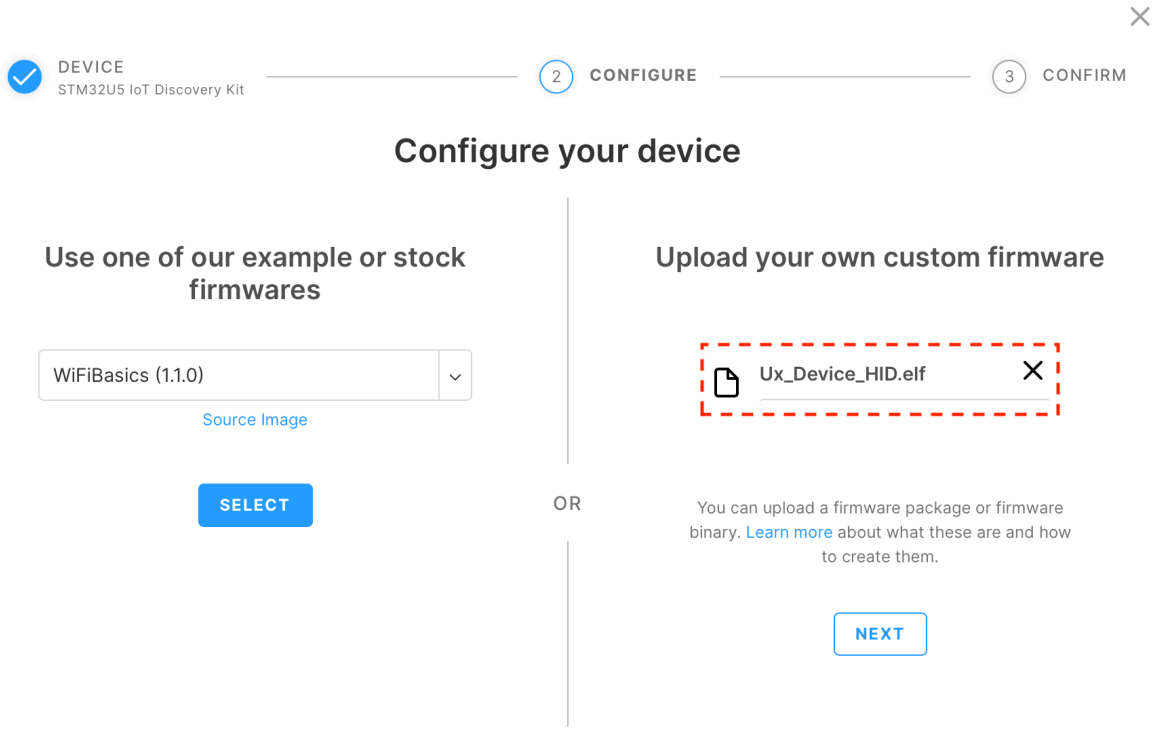
9. The firmware file is ready to be uploaded to your AVH virtual STM32U5 board.

2.8 Attach a Virtual USB Peripheral to a Virtual Raspberry Pi 4 Board

To attach a virtual joystick to the Raspberry Pi 4:

1. Create a new virtual STM32U5 board and manually upload the `Ux_Device_HID.elf` custom firmware you created.

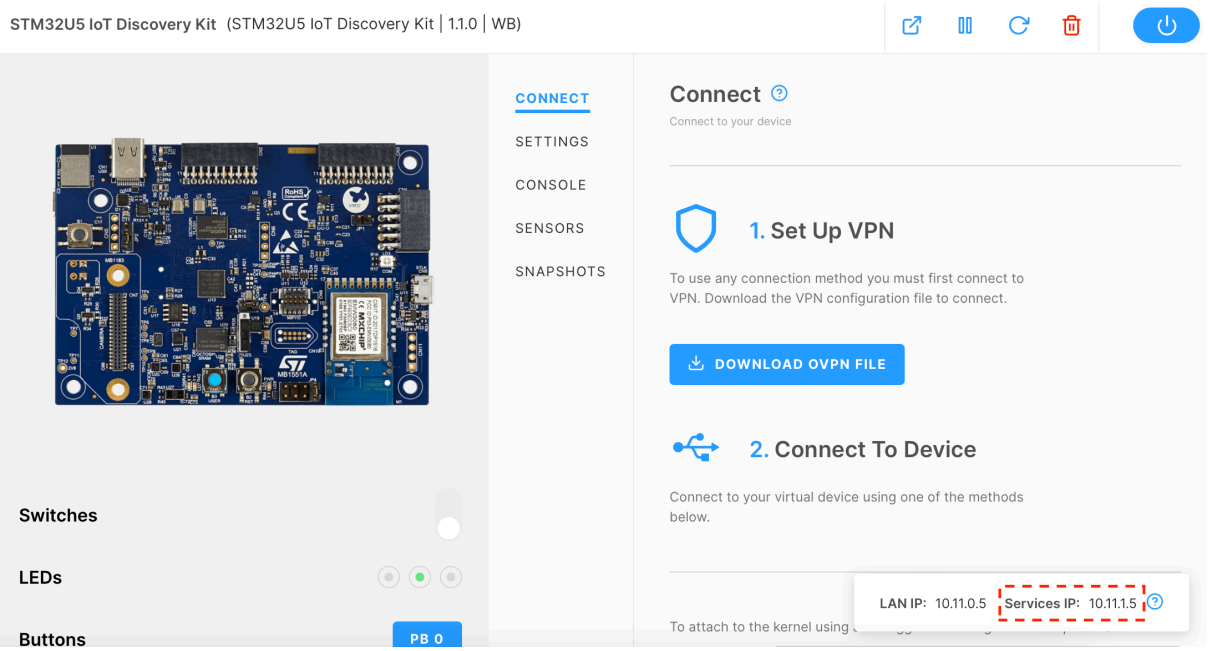
Figure 2-22: Upload joystick firmware



Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2. Make note of the Services IP address on the Connect tab. In our example, we will use the Services IP `10.11.1.5`.

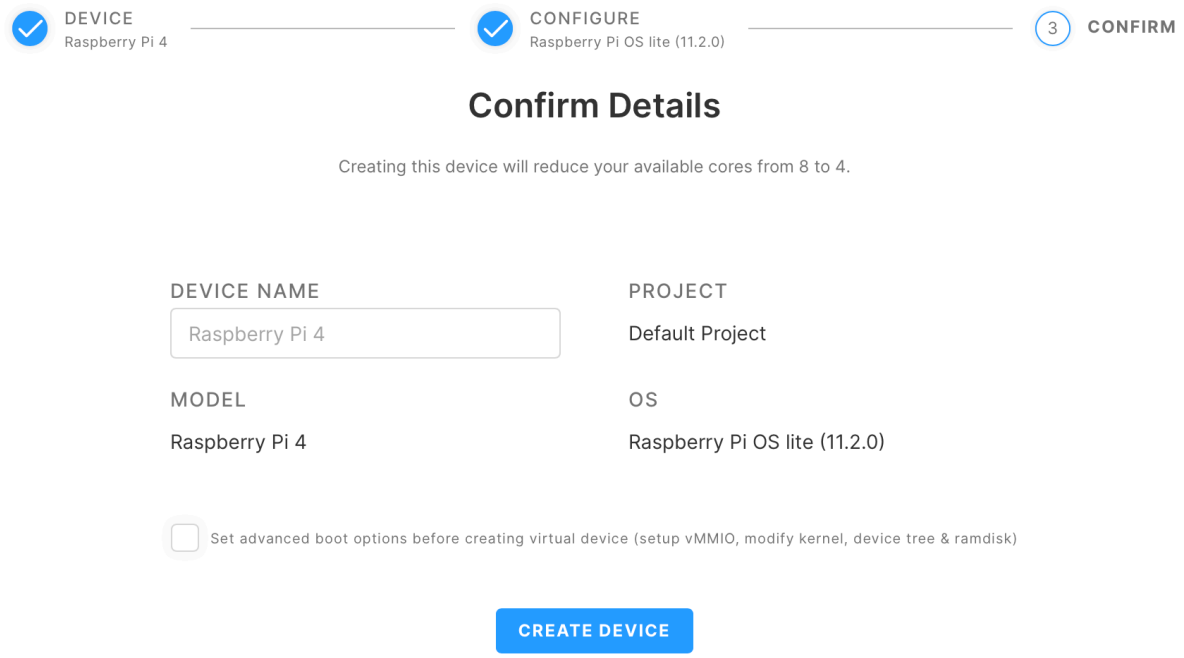
Figure 2-23: Joystick Services IP



3. Create a new Raspberry Pi 4 virtual device using any version of the firmware. In our example, we will use the lite version of firmware `11.2.0`.

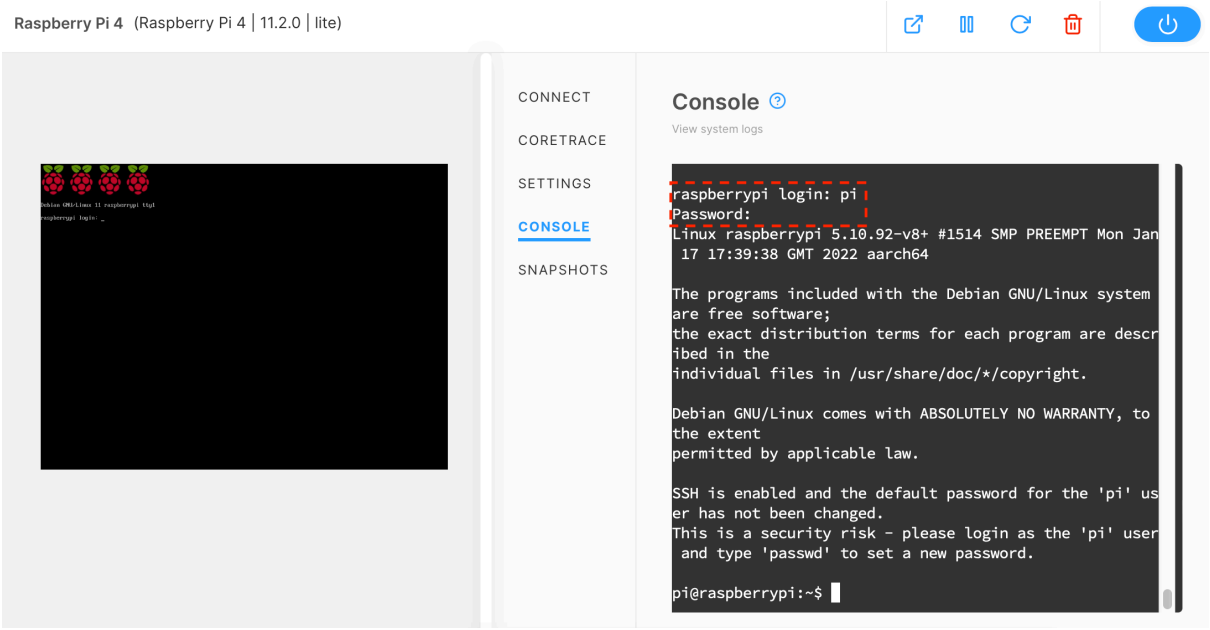
Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-24: Create new Pi



4. Log in as the root user with the username `pi` and password `raspberry`.

Figure 2-25: Login to Pi as root

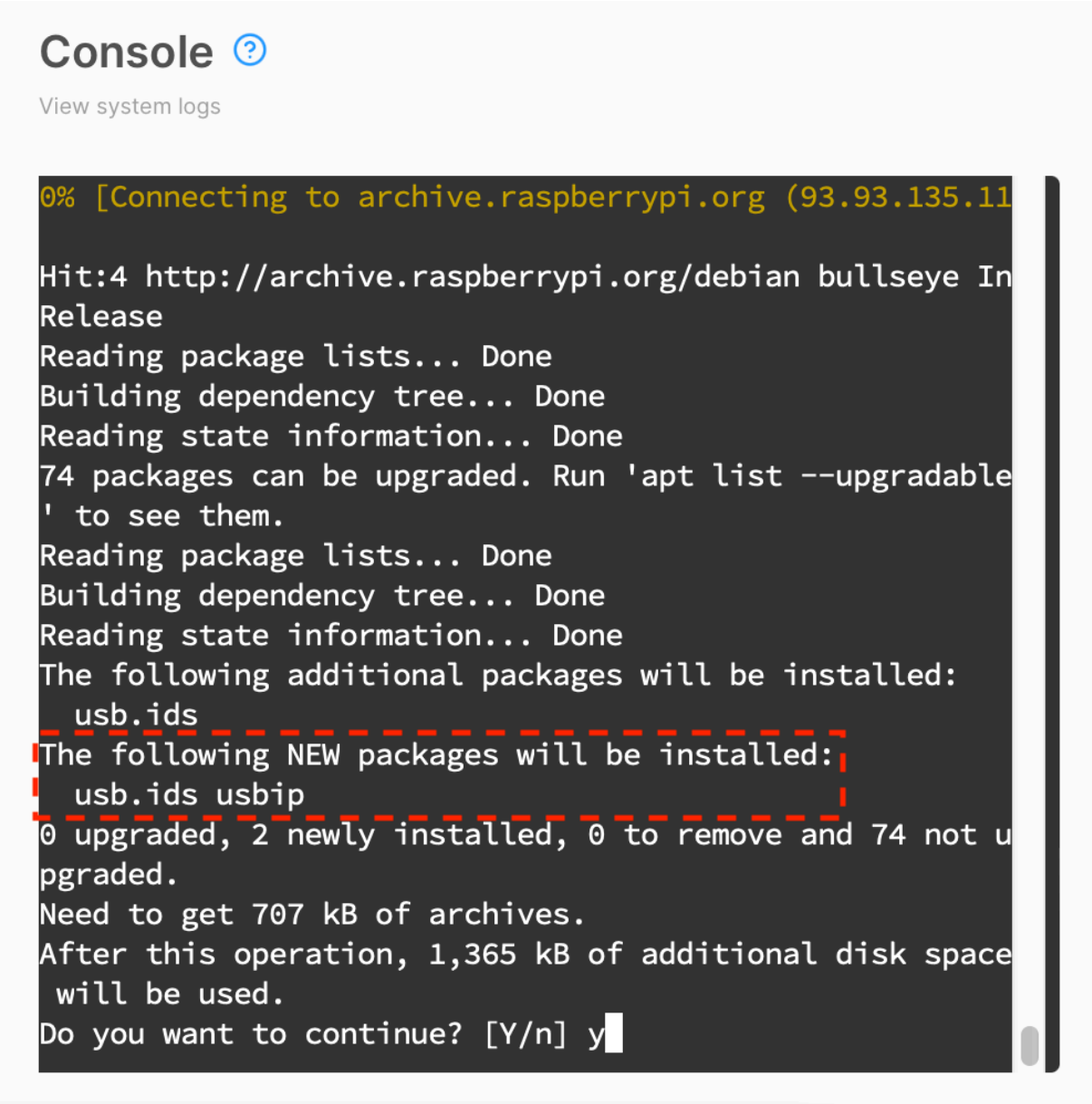


Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5. Install the USB/IP package.

```
sudo apt update && sudo apt install -y usbip
```

Figure 2-26: Install usbip

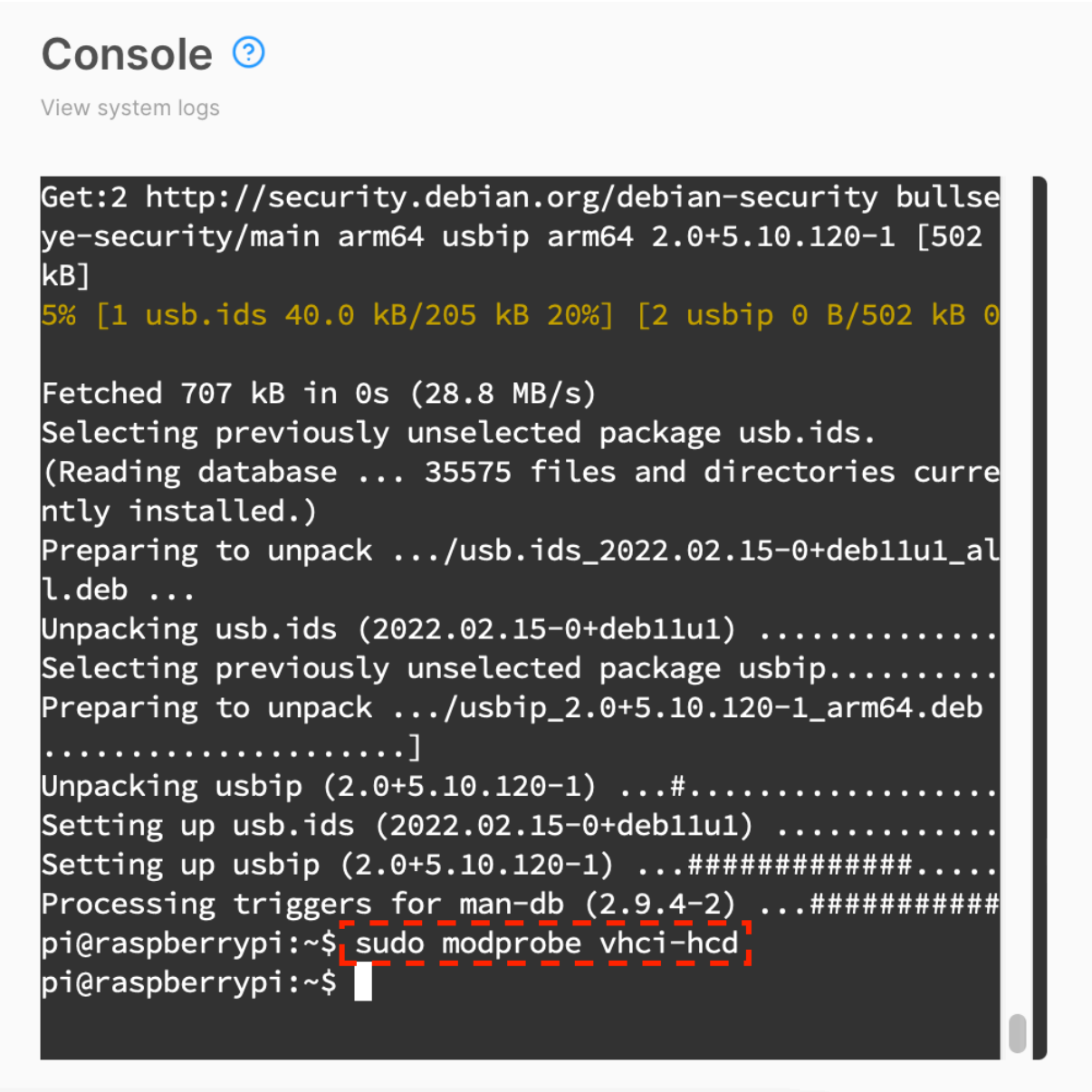


6. Load the Virtual Host Controller Interface kernel module.

```
sudo modprobe vhci-hcd
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-27: Load kernel module

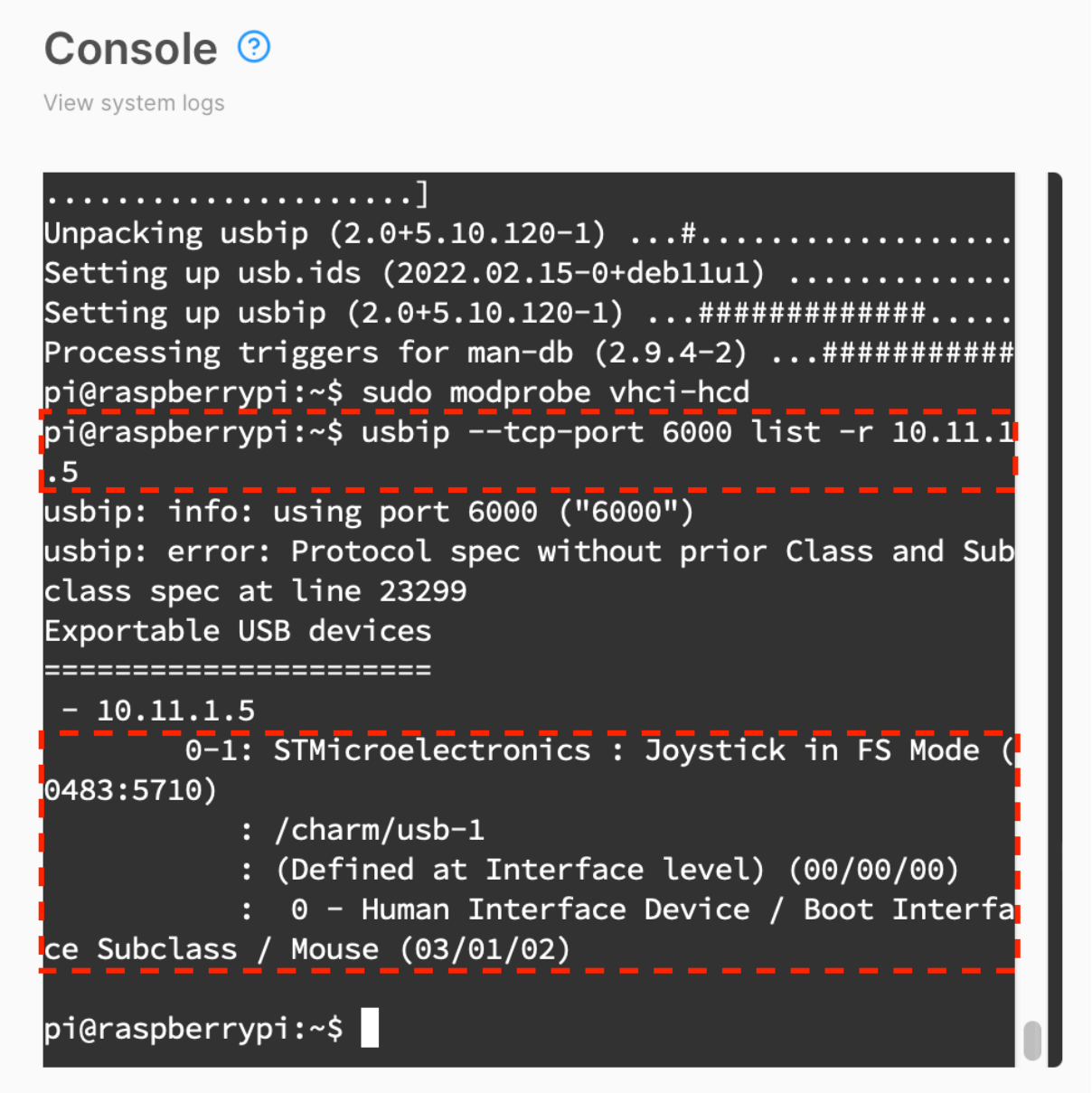


7. List the STM32U5's exportable USB devices using the Services IP and port 6000.

```
usbip --tcp-port 6000 list -r 10.11.1.5
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-28: List USB devices

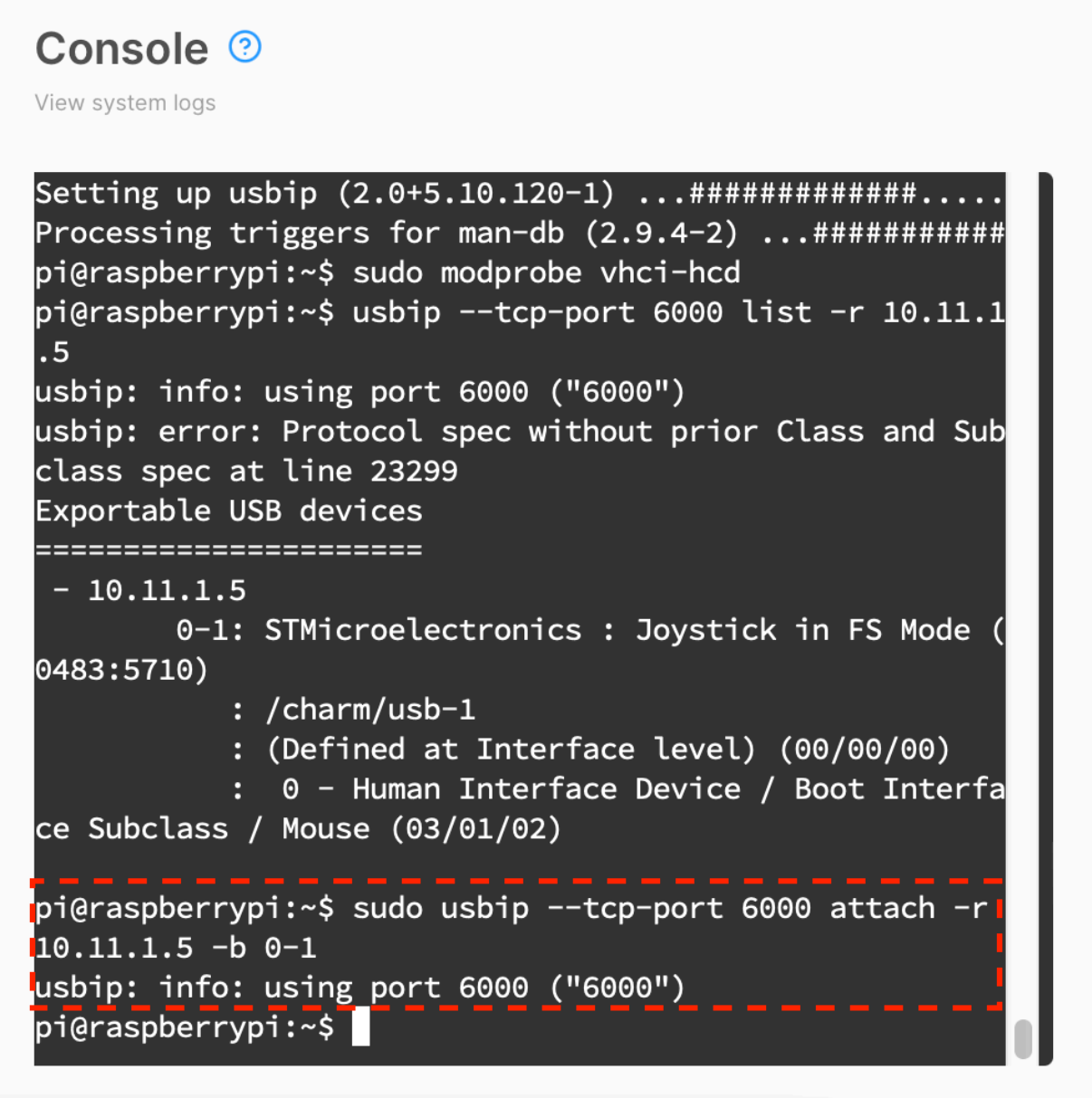


8. Attach to USB bus 0-1, which corresponds to the joystick peripheral. (Don't forget to `sudo` or the command will fail.)

```
sudo usbip --tcp-port 6000 attach -r 10.11.3.3 -b 1-2
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

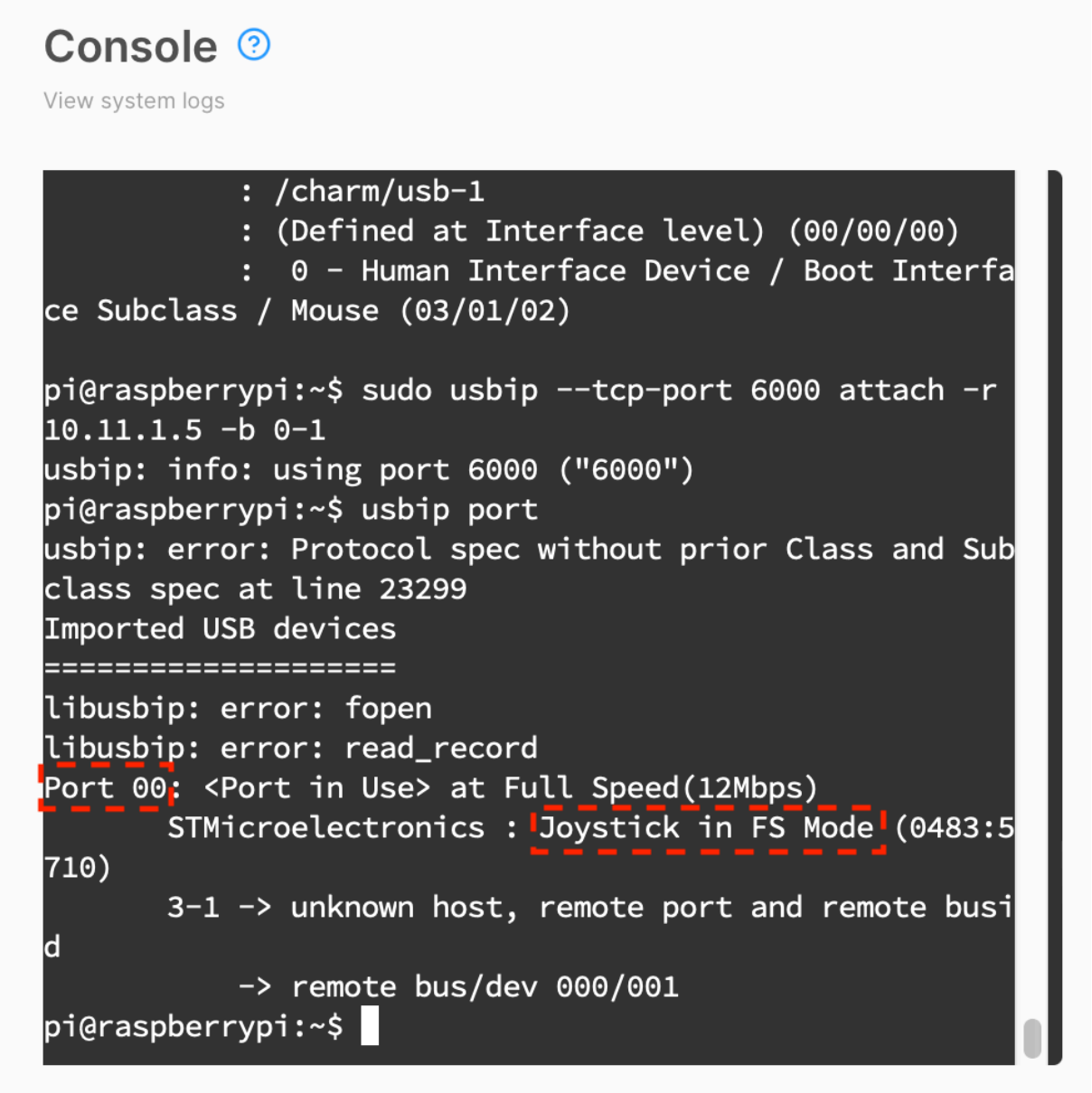
Figure 2-29: Attach to USB bus 0-1



9. The STM32U5's virtual joystick is now attached to your Raspberry Pi 4 virtual board. Confirm the device is attached using `usbip port`. Make note that the device is using port `00`.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-30: Confirm the device is attached

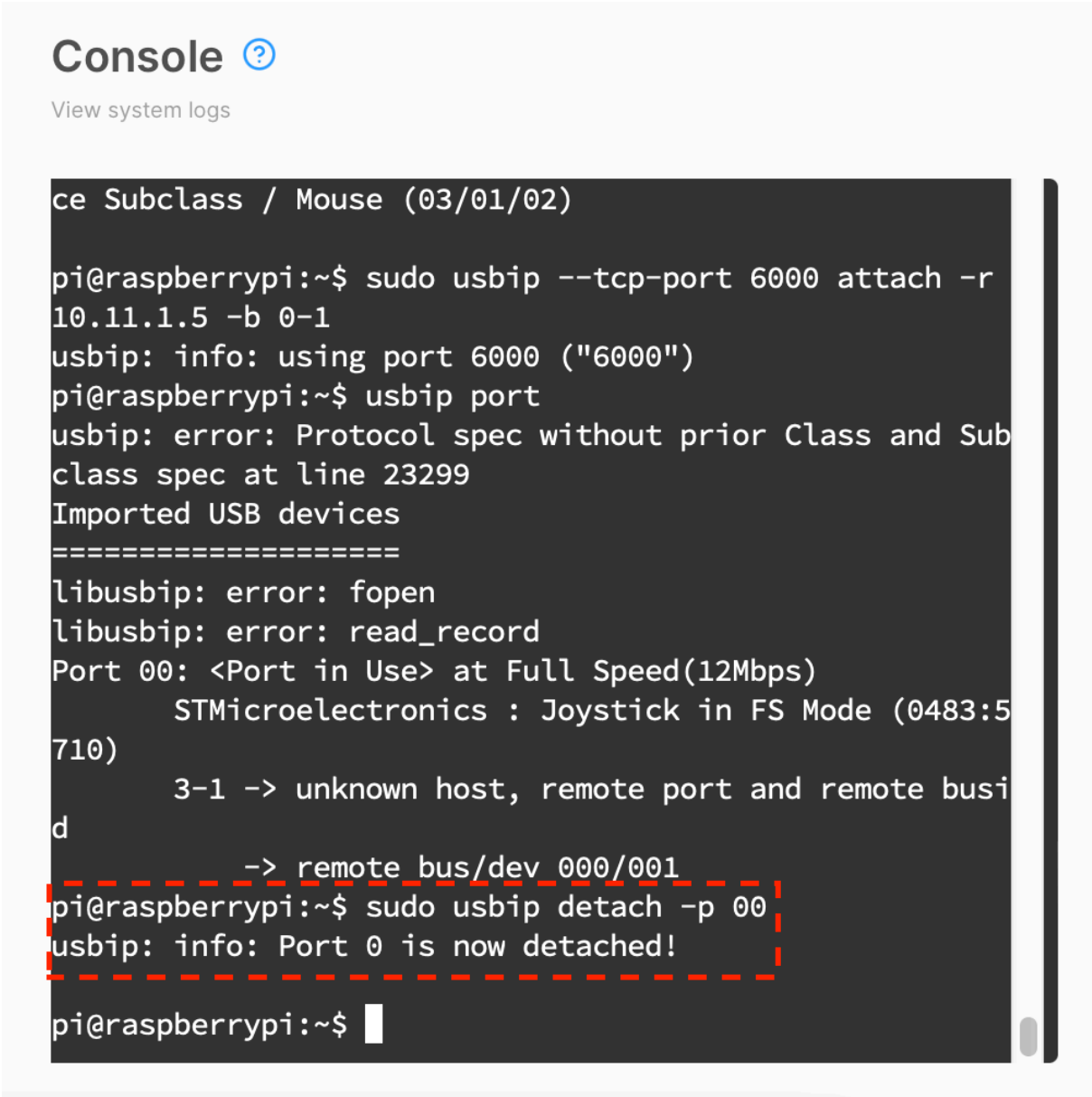


10. When you are finished, detach the peripheral from port `00`.

```
sudo usbip detach -p 00
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-31: Detach the peripheral



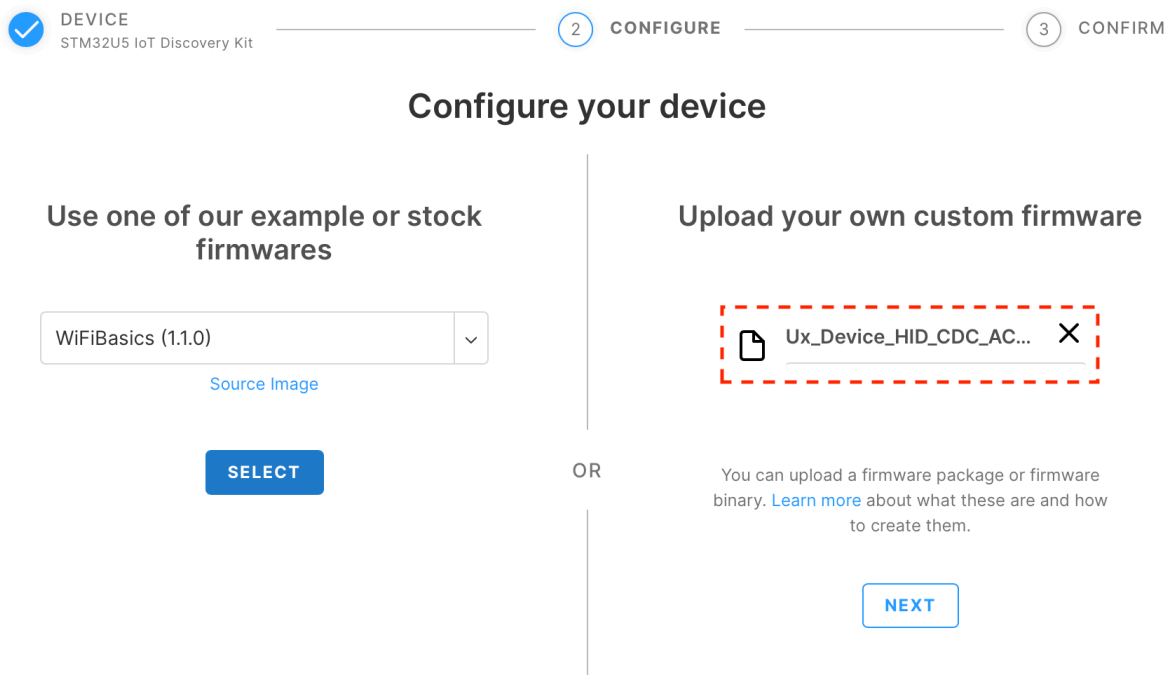
Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.9 Attach a Virtual USB Peripheral to a Physical Computer Using a VPN Connection

In this example, we will use a laptop running Ubuntu 20.04. We will also use Ux_Device_HID_CDC_ACM.elf, the "Mini LED Display" firmware file we created in the Build USB Firmware Files with the STM32CubeIDE Tool step.

1. Create a new STM32U5 virtual device and manually upload the `Ux_Device_HID_CDC_ACM.elf` firmware you created in the STM32CubeIDE tool.

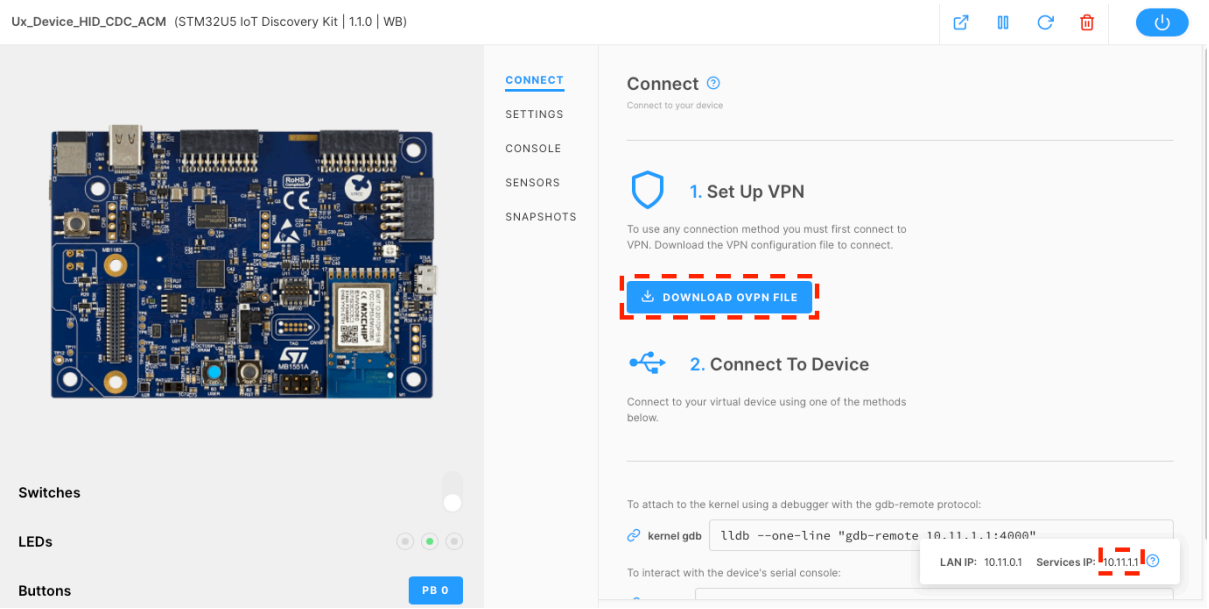
   Figure 2-32: Upload LED firmware



2. Download the `.ovpn` file and make note of the Services IP address on the Connect tab. In our example, the Services IP is `10.11.1.1`.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-33: STM32U5 VPN and Services IP



3. On your Linux client machine, install the required packages.

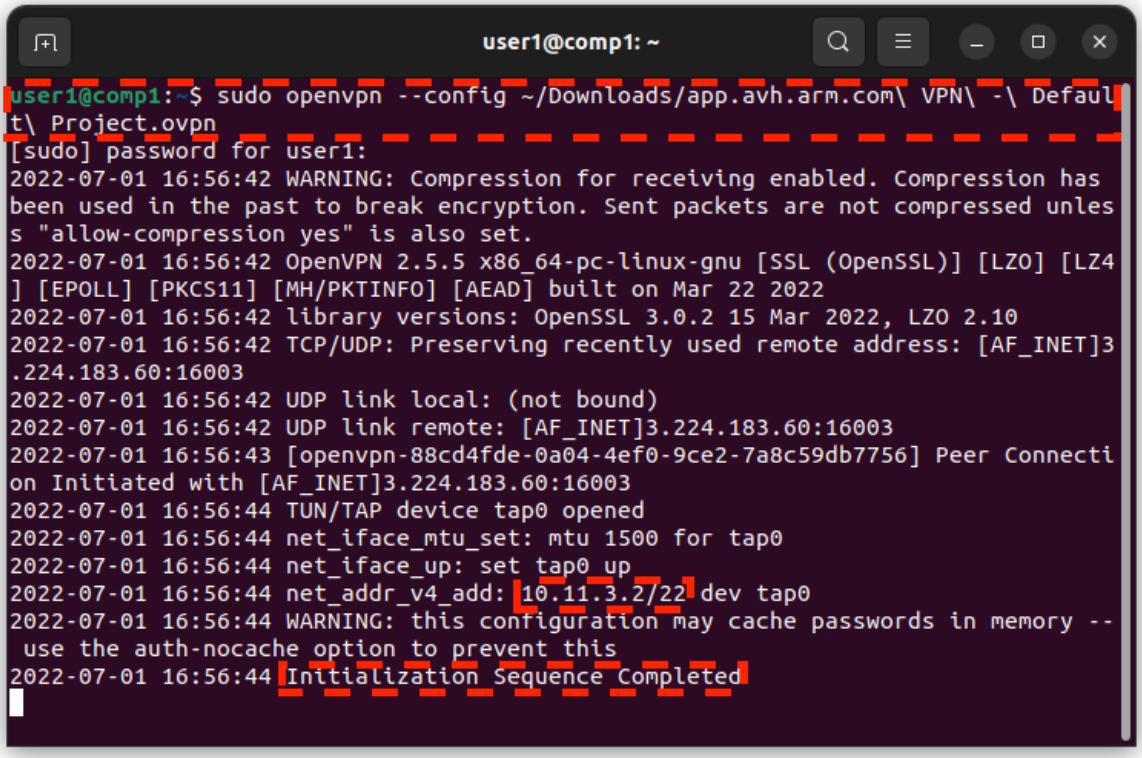
```
sudo apt update && sudo apt install -y linux-tools-$(uname -r) hwdata
```

4. Connect to the AVH VPN using the `.ovpn` file you downloaded.

```
sudo openvpn --config /Downloads/app.avh.arm.com\ VPN\ -\ Default\ Project.ovpn
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

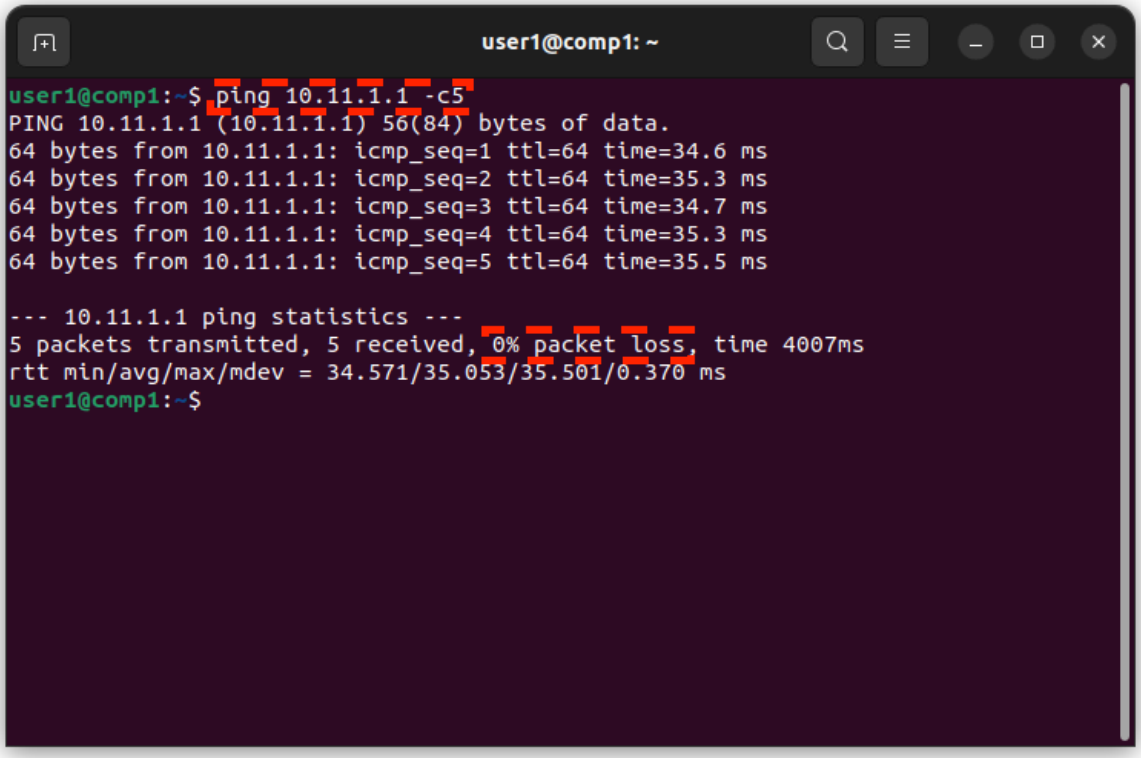
Figure 2-34: OpenVPN



5. Open a second Terminal window. Double-check your VPN connection by pinging the Services IP for your USB/IP host.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-35: Ping Test

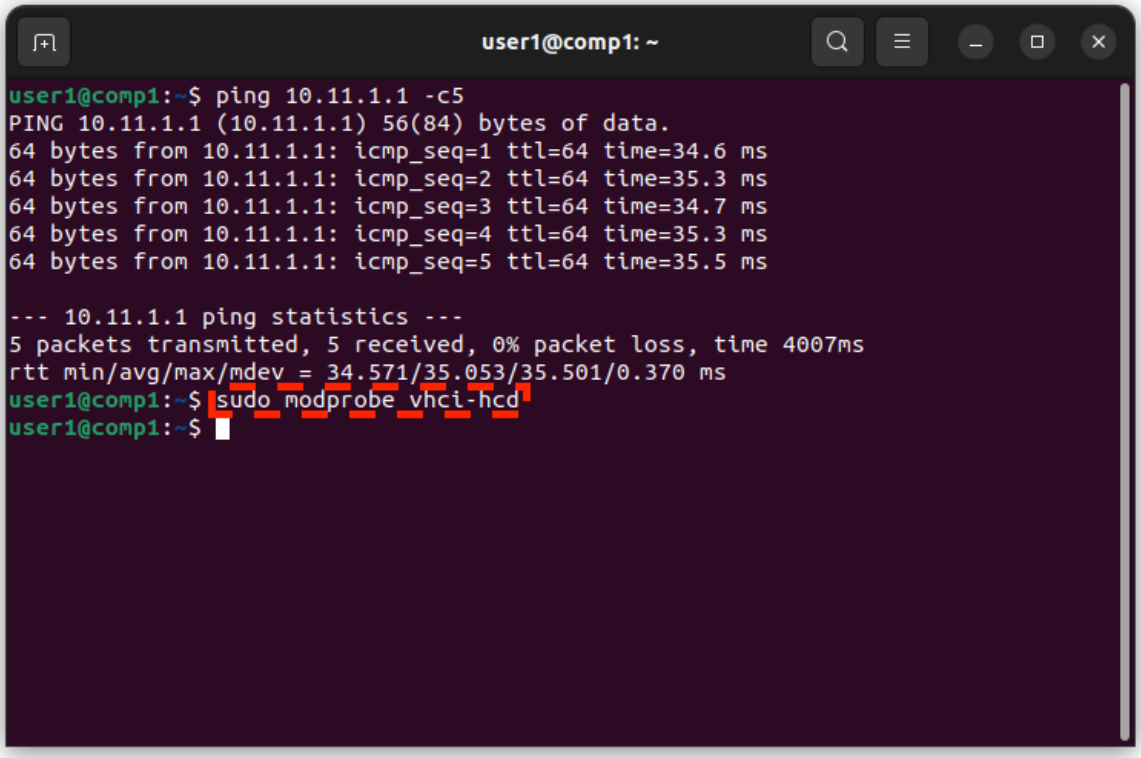


6. Load the Virtual Host Controller Interface kernel module.

```
sudo modprobe vhci-hcd
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-36: Load the kernel module

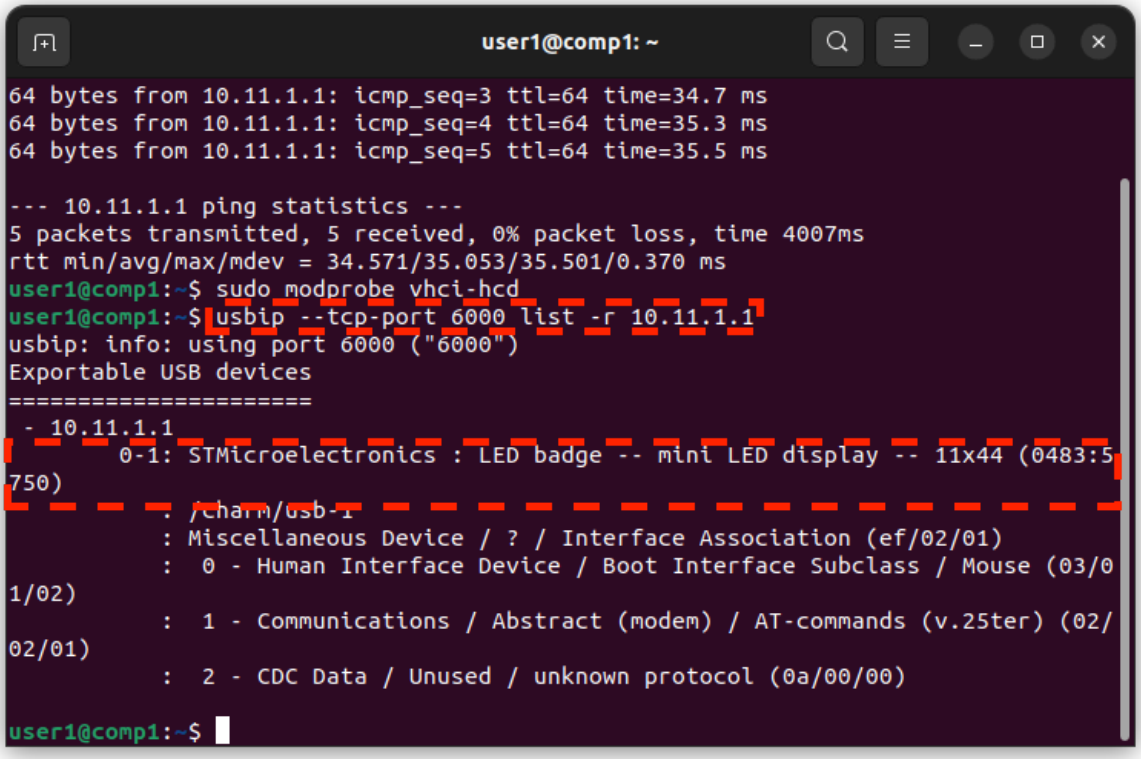


7. List the exportable STM32U5 USB devices.

```
usbip --tcp-port 6000 list -r 10.11.1.1
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-37: List USB Devices



8. Attach to USB bus 0-1, which corresponds to the LED display.

```
sudo usbip --tcp-port 6000 attach -r 10.11.1.1 -b 0-1
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-38: Attach to USB Device

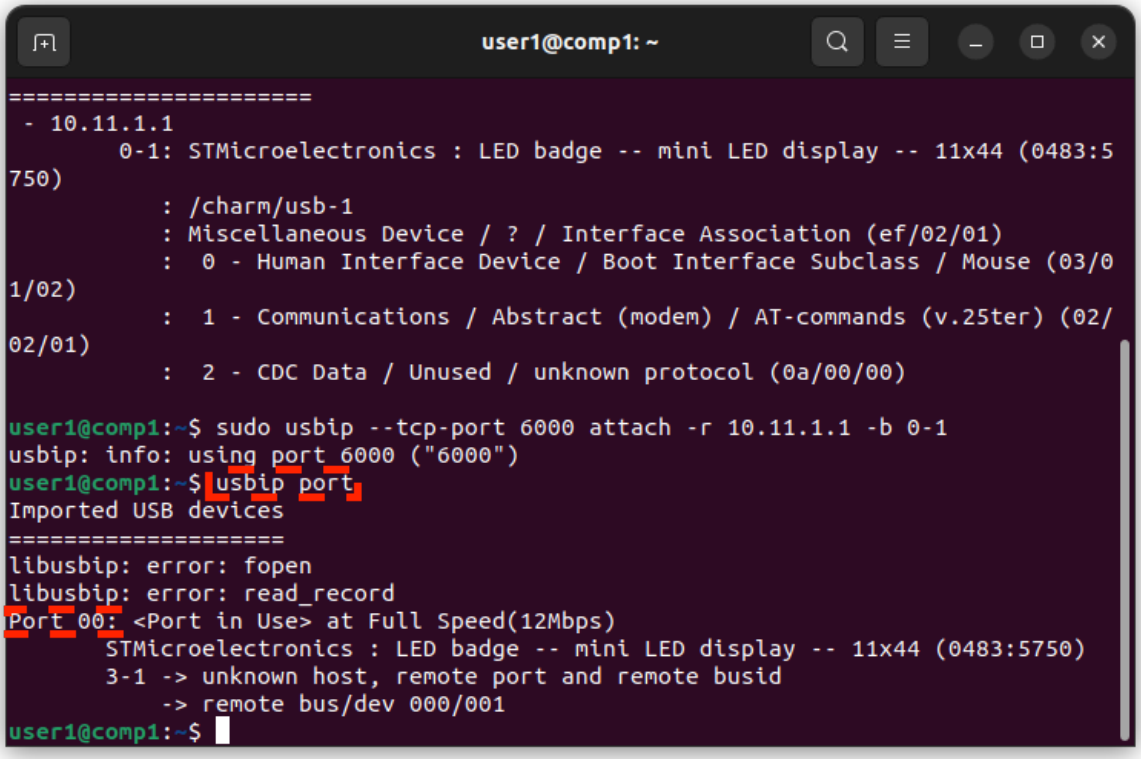
```
user1@comp1: ~
64 bytes from 10.11.1.1: icmp_seq=5 ttl=64 time=34.7 ms

--- 10.11.1.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 34.660/36.928/40.865/2.114 ms
user1@comp1:~$ sudo modprobe vhci-hcd
user1@comp1:~$ usbip --tcp-port 6000 list -r 10.11.1.1
usbip: info: using port 6000 ("6000")
Exportable USB devices
======================
 - 10.11.1.1
        0-1: STMicroelectronics : LED badge -- mini LED display -- 11x44 (0483:5
750)
           : /charm/usb-1
           : Miscellaneous Device / ? / Interface Association (ef/02/01)
           :  0 - Human Interface Device / Boot Interface Subclass / Mouse (03/0
1/02)
           :  1 - Communications / Abstract (modem) / AT-commands (v.25ter) (02/
02/01)
           :  2 - CDC Data / Unused / unknown protocol (0a/00/00)

user1@comp1:~$ sudo usbip --tcp-port 6000 attach -r 10.11.1.1 -b 0-1
usbip: info: using port 6000 ("6000")
user1@comp1:~$
```

9. List the Imported USB devices to confirm the attach command was successful.

```
usbip port
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-39: Imported USB Device

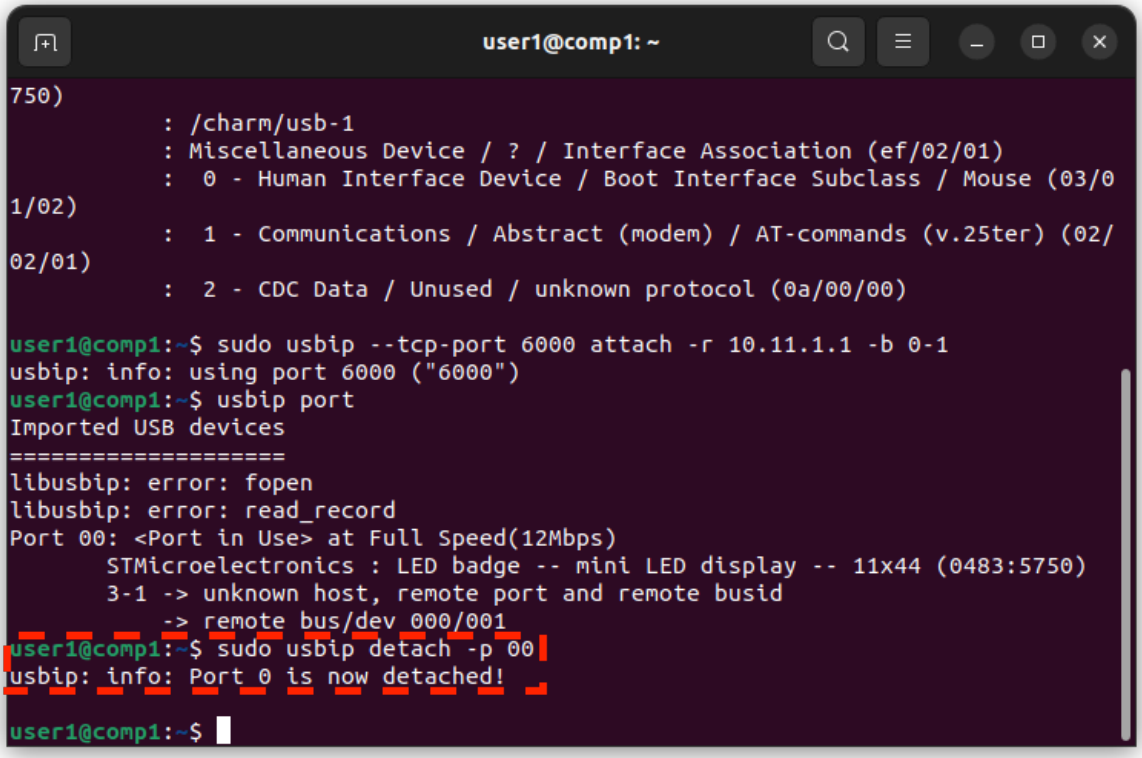


10. When you are finished, detach from port `00`.

```
sudo usbip detach -p 00
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-40: Detach USB Device



2.10 Attach a Virtual USB Peripheral to a Physical Computer Using SSH Port Forwarding

In this example, we will use a laptop running Ubuntu 22.04. We will also use `Ux_Device_HID_CDC.elf`, the "Joystick in FS Mode" firmware file we created in the Build USB Firmware Files with the STM32CubeIDE Tool step.

1. Upload the `Ux_Device_HID_CDC.elf` firmware file and start the STM device. Make note of the device's "Services IP".
2. Next, we will find your project's unique identifier. The first option is to use the Get Projects API command. Alternatively, you can do the following.
   a. Boot up a virtual Raspberry Pi 4 board.
   b. Under the CONNECT tab, find to the Quick Connect SSH command
   c. Copy the part of the command showing your project's unique identifier `<project identifier>@proxy.app.avh.arm.com`.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3. On your Ubuntu computer, generate a public/private key pair using the Ed25519 signature scheme. In our example, we will use the default file path with no passphrase.

```
ssh-keygen -t ed25519
```

Add the public key (`cat ~/.ssh/id_ed25519.pub`) to your AVH project's authorized project keys under https://app.avh.arm.com/admin/projects.

1. Install USB/IP and the hardware identification tool.

```
sudo apt update && sudo apt install -y linux-tools-$(uname -r) hwdata
```

2. Load the USB Virtual Host Controller Interface kernel module.

```
sudo modprobe vhci-hcd
```

3. Use ssh to forward local port 6000 to port 6000 of your device's services IP.

```
ssh -M -Ssock -N -f -L 6000:<STM_device_services_IP>:6000
 <your_project_id>@proxy.app.avh.arm.com
```

4. List USBIP devices on localhost port 6000.

```
usbip --tcp-port 6000 list -r localhost
```

5. You should see the virtual STMicroelectronics USB peripheral appear.
6. When you are finished, close the socket.

```
ssh -Ssock -O exit proxy.corellium.com
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3. vMMIO Introduction

Getting started with virtual MMIO (vMMIO):

- Background
- The CoreIO Library API
  - Connecting to a target VM
  - Registering a vMMIO Range
  - Registering a vMMIO Handler
  - Processing Read/Write Events
  - Interrupts
  - Direct Memory Access (DMA)
- Examples

3.1 Background

On physical devices, software and hardware interact in a predefined way to yield predefined results. Namely, hardware is driven by software drivers according to its specification. Fundamentally, this interaction distills into a series of address and data bus read and write operations. On AVH virtual devices this behavior is identical, save for the distinction that software interacts with virtual software-defined models of the physical hardware device being virtualized.

AVH implements and extends virtual hardware model capabilities through what is referred to as Virtual Memory Mapped IO (vMMIO). This allows user-supplied code to be executed in response to VM physical memory read or write bus transactions. The full suite of features available also include DMA for arbitrary memory read and write, as well as control of a dedicated virtual IRQ line.

3.2 The CoreIO Library API

The CoreIO library allows external programs to act as MMIO responders, send interrupt requests to the VM, and perform DMA.

3.3 Connecting to a target VM

To use the library in your C program, you need to connect to a VM:

```
int coreio_connect(const char *target);
      Connect to a VM.
        target          string like "10.10.0.3:23456"
      Returns error flag.
```

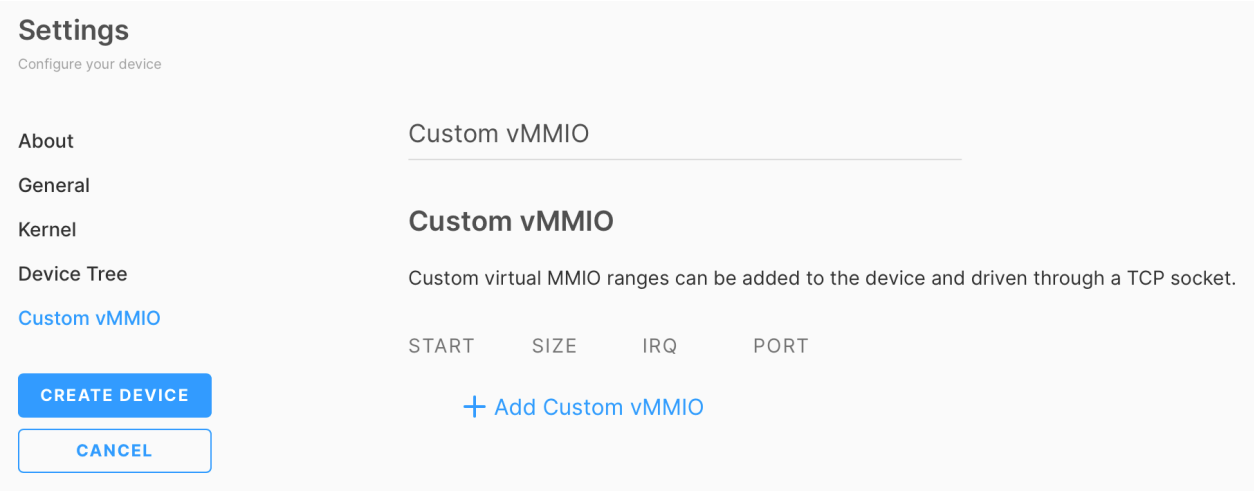
Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If the result was zero, you can continue using the CoreIO API.

3.4 Registering a vMMIO Range

The vMMIO ranges must be created either through the web API or through the UI. After VM creation, a vMMIO range may be created by using the relevant tab under `Settings`.

Figure 3-1: Custom vMMIO



`Start` and `Size` represent the physical address space of the virtual MMIO range being created. The `IRQ` is used when vMMIO handlers implement a virtual IRQ line linked to the core interrupt controller. And `port` represents the network port on which AVH will listen for connections.

Note

vMMIO ranges created without any associated client handlers (below) will not trigger bus faults within the VM. The default behavior is to return a `0xaa` in each byte up to the width size of the access (e.g. `0xaaaa` for two byte reads).

3.5 Registering a vMMIO Handler

After registering the range, the access handler must also be registered so that AVH knows where to route the access events to.

The API is based around a few concepts. First, there's the vMMIO range. Each range is a 4k-aligned block of address space that a CoreIO client can claim. Ranges are indexed from 0, in the order they were specified during VM creation. Up to 16 vMMIO ranges can be created per VM instance.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

To claim a vMMIO range, create and fill out a callback structure:

```
#define COREIO_FLAGS_IFETCH     2
    #define COREIO_FLAGS_PRIV       4
    #define COREIO_FLAGS_SECURE     8

    typedef struct {
        /* MMIO operations. */
        int (*read)(void *priv, uint64_t addr, size_t len, void *buf, unsigned
 flags);
        int (*write)(void *priv, uint64_t addr, size_t len, void *buf, unsigned
 flags);
        /* MMIO pair-wise operations. If not supplied, read/write are used twice. */
        int (*readp)(void *priv, uint64_t addr, size_t len, void *buf1, void *buf2,
 unsigned flags);
        int (*writep)(void *priv, uint64_t addr, size_t len, void *buf1, void *buf2,
 unsigned flags);
    } coreio_func_t;
```

The read/write functions are called with:

- `priv` - your unmodified private pointer (`funcp` from register function)
- `addr` - VM physical address of access
- `len` - Byte length of access, for example 1, 2, 4, 8, 16 ...
- `buf` - Data buffer (contains VM write contents on write, else destinations buffer for reads)
- `flags` - Flags describing access mode, see `COREIO_FLAGS_*`

The functions can return 1 to signal failure, which will become a hardware-origin bus error when it hits the VM.

After you have your handlers and `coreio_func_t` filled out, register it:

```
int coreio_register(unsigned rid, const coreio_func_t *func, void *funcp);
      Register a MMIO handler.
        rid             MMIO range ID to register
        func            io handler function table
        funcp           io handler function parameter (passed to callbacks)
      Returns error flag.
```

3.6 Processing Read/Write Events

Then you must construct a main loop of your program. The main loop support is intended to work with `select()`. You call `coreio_preparefds` before `select()` to add CoreIO's sockets to select wait list, and `coreio_processfds` after select to process the socket events:

```
int coreio_preparefds(int nfds, fd_set *readfds, fd_set *writefds);
      Prepare fd_sets for select(2).
        nfds            current index of maximum fd in sets + 1
        readfds         readfds to update
        writefds        writefds to update
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
      Returns new nfds.
```

```
int coreio_processfds(fd_set *readfds, fd_set *writefds);
      Process fd_sets after select(2).
        readfds         readfds to process
        writefds        writefds to process
      Returns error flag (for instance, connection to VM lost).
```

A simple implementation of a main loop, without any other event processing, looks like this:

```
/* ... */
    fd_set readfds, writefds;
    int res;

    while(1) {
        FD_ZERO(&writefds);
        FD_ZERO(&readfds);
        int nfds = coreio_preparefds(0, &readfds, &writefds);
        if(nfds < 0) /* error in prepare */
            break;
        select(nfds, &readfds, &writefds, NULL, NULL);
        res = coreio_processfds(&readfds, &writefds);
        if(res) /* error in process */
            break;
    }
    /* ... */
```

This kind of main loop, with a timeout, is provided as a convenience by the library:

```
int coreio_mainloop(long long usec);
      Simple implementation of a main loop.
        usec            time to spend in loop, in microseconds;
                        negative means forever
      Returns error flag.
```

3.7 Interrupts

In addition to processing MMIO events, your program can also send interrupts. Note that in the VMs, all interrupt lines are level. The interrupt controller can interpret them as edge, but to trigger an edge interrupt you must send two updates: one from 0 to 1, one from 1 to 0. It is not sufficient to send 1 on each IRQ trigger as this does not create an edge event.

```
int coreio_irq_update(unsigned rid, unsigned iid, unsigned state);
      Send an IRQ update.
        rid             MMIO range ID associated with IRQ
        iid             IRQ index
        state           IRQ line state (1 - active, 0 - inactive)
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.8 Direct Memory Access (DMA)

There are two ways to perform DMA access, blocking and non-blocking. Blocking access is simpler. You just call a read/write function, and it returns when done. Non-blocking access requires a completion function (and its opaque parameter pointer, which will be passed to it without change). The function will be called at a future time during `coreio_processfds()`.

```
int coreio_dma_read(unsigned mid, unsigned sid, uint64_t addr,
        size_t len,void *buf, unsigned flags,
        void (*cpl)(void *, int), void *cplp);

      Start a DMA read.
        mid             IOMMU ID (0 = direct)
        sid             IOMMU region ID
        addr            target address
        size            size of read to perform
        buf             memory buffer to fill with data
        flags           bus access flags
        cpl             completion function; if NULL, call is blocking
        cplp            completion function parameter
      Returns error flag.
```

```
int coreio_dma_write(unsigned mid, unsigned sid, uint64_t addr,
        size_t len, void *buf, unsigned flags,
        void (*cpl)(void *, int), void *cplp);

      Start a DMA write.
        mid             IOMMU ID (0 = direct)
        sid             IOMMU region ID
        addr            target address
        size            size of write to perform
        buf             memory buffer with data
        flags           bus access flags
        cpl             completion function; if NULL, call is blocking
        cplp            completion function parameter
      Returns error flag.
```

On most small VMs, you can pass 0 as IOMMU ID and region ID. This specifies that the access should target VM physical memory directly.

Finally, to cleanly shut down the CoreIO library, use:

```
void coreio_disconnect(void);
```

3.9 Examples

The example sources include

`rpi-test`

- Use an inside and outside VM agent for demonstrating end-to-end vMMIO cycle

`slider-mmio`

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Use Python bindings to send an IRQ to the VM depending on the state of a graphical slider

`vsi-waveio`

- Create a whole virtual device model confronting to Arm's VSI specification that streams a WAV file into the VM

Next steps Raspberry Pi vMMIO Example

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4. Kernel debugging

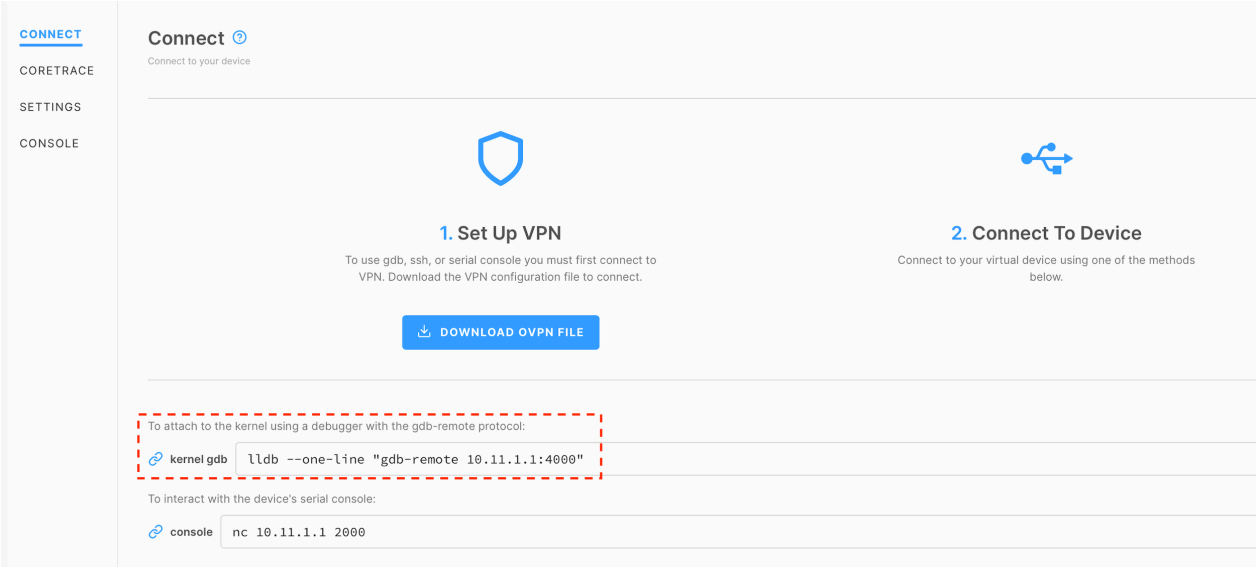
This article shows you how to debug a VM kernel.

4.1 Using GDB/LLDB

AVH implements a GDB server stub to support VM debugging. This allows any debugger that understands this protocol to be used as well, including `lldb`.

After creating a VM, the `Connect` tab provides an `lldb` one-liner that can be used to establish a debugging session with the VM.

Figure 4-1: Connect



While `lldb` is specified in the UI, users are free to use `gdb` as well

```
$ gdb
(gdb) target remote 10.11.1.1:4000
Remote debugging using 10.11.1.1:4000
0xffff80001117094c in ?? ()
(gdb) info reg
x0             0xe0     224
x1             0x11c430 1164336
x2             0x0      0
x3             0x4000000000000000       4611686018427387904
x4             0x0      0
x5             0xffff8000dde56000       -140733765558272
x6             0x80     128
...
(gdb) x/10i $pc
=> 0xffff80001117094c:  hint    #0x1d
   0xffff800011170950:  ret
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
   0xffff800011170954:  hint    #0x19
   0xffff800011170958:  stp     x29, x30, [sp,#-16]!
   0xffff80001117095c:  mov     x29, sp
   0xffff800011170960:  bl      0xffff800011170940
   0xffff800011170964:  ldp     x29, x30, [sp],#16
   0xffff800011170968:  hint    #0x1d
   0xffff80001117096c:  ret
   0xffff800011170970:  hint    #0x19
```

All the usual debugger features work, including single-step, breakpoints, watchpoints, memory inspection, register inspection, register or memory editing, etc. Consult a guide specific to your debugger.

4.2 Monitor Commands

The so-called monitor commands are raw packets interpreted directly by the debug server and not by the client (or marshaled into any known command format). In other words, the monitor is convenient way to expand a debugger's functionality without changing the client code. `gdb` exposes this functionality by prefixing your command with `monitor`. Sending e.g. `monitor help` requests that AVH respond with its list of available commands

```
(gdb) monitor help
Available monitor commands:
  help                  List all commands.
  info                  Show VM information.
  sr                    Access CPU system registers.
  pt                    Dump current pagetable.
  v2p                   Translate a virtual address to physical.
  p2v                   Translate a physical address to virtual.
  vcont-dflt            Apply default vCont action if specific actions are present.
  phys                  Switch to physical access mode (optionally takes base
 address).
  virt                  Switch to virtual access mode.
  bt                    Show iOS-style backtrace.
  thread                Show current thread.
```

To list the contents of the system registers one may use `monitor sr`.

```
(gdb) monitor sr
System registers of CPU 0:
  sctlr_el1        = 0x0000000030d4d91d
  cpacr_el1        = 0x0000000000300000
  zcr_el1          = 0x0000000000000000
  ttbr0_el1        = 0x00000001061c5000
  ttbr1_el1        = 0x0188000041859000
  tcr_el1          = 0x00000032b5503510
...
```

Or list the current configuration of the page tables of the current virtual CPU (vCPU).

```
(gdb) monitor pt
Pagetable for CPU 0:
  0000004000000000-0000004000000fff -> 010a022000 local user pXN uXN memory Owb-rwa
 Iwb-rwa
```

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
  0000004000001000-0000004000001fff -> 010a0d9000 local user pXN uXN memory Owb-rwa
 Iwb-rwa
  0000004000002000-0000004000002fff -> 010a023000 local user pXN uXN memory Owb-rwa
 Iwb-rwa
  0000004000008000-0000004000008fff -> 0108871000 local user pXN uXN memory Owb-rwa
 Iwb-rwa
  0000004000009000-0000004000009fff -> 0108741000 local user pXN uXN memory Owb-rwa
 Iwb-rwa
  000000400000a000-000000400000afff -> 010a1db000 local user pXN uXN memory Owb-rwa
 Iwb-rwa
...
```

`lldb` can be used similarly to send monitor commands using a slightly more verbose command structure.

```
$ lldb --one-line "gdb-remote 10.11.1.1:4000"
(lldb) gdb-remote 10.11.1.1:4000
Process 1 stopped
* thread #1, stop reason = signal SIGINT
    frame #0: 0xffffffe01117094c
Target 0: (No executable module.) stopped.
(lldb) process plugin packet monitor help
Available monitor commands:
  help                  List all commands.
  info                  Show VM information.
  sr                    Access CPU system registers.
...
```

Note

Debugger threads are discrete VM CPUs, and so switching between threads allows users to interact with specific cores.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

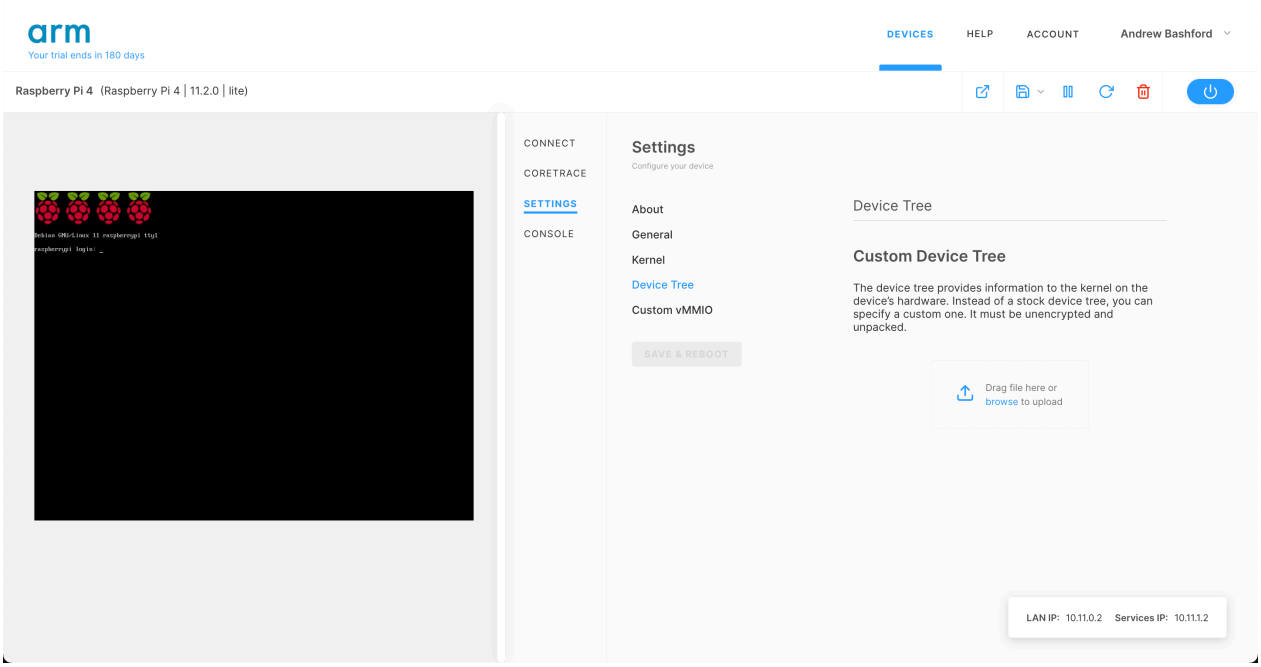
5. Custom device tree

Custom Device Trees are available on Raspberry Pi 4 and the iMX8m Arm Cortex Complex devices.

A device tree is a representation of hardware used by boot loaders provided to the kernel. It is a hierarchical representation of devices by connection, used by the kernel to communicate with the I/O buses and other low-level hardware.

Instead of a stock device tree, you can specify a custom one. It must be unencrypted and unpacked. To upload a custom device tree, simply head to Settings > Device Tree and drag-and-drop your file to upload it.

Figure 5-1: Custom device tree



Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential