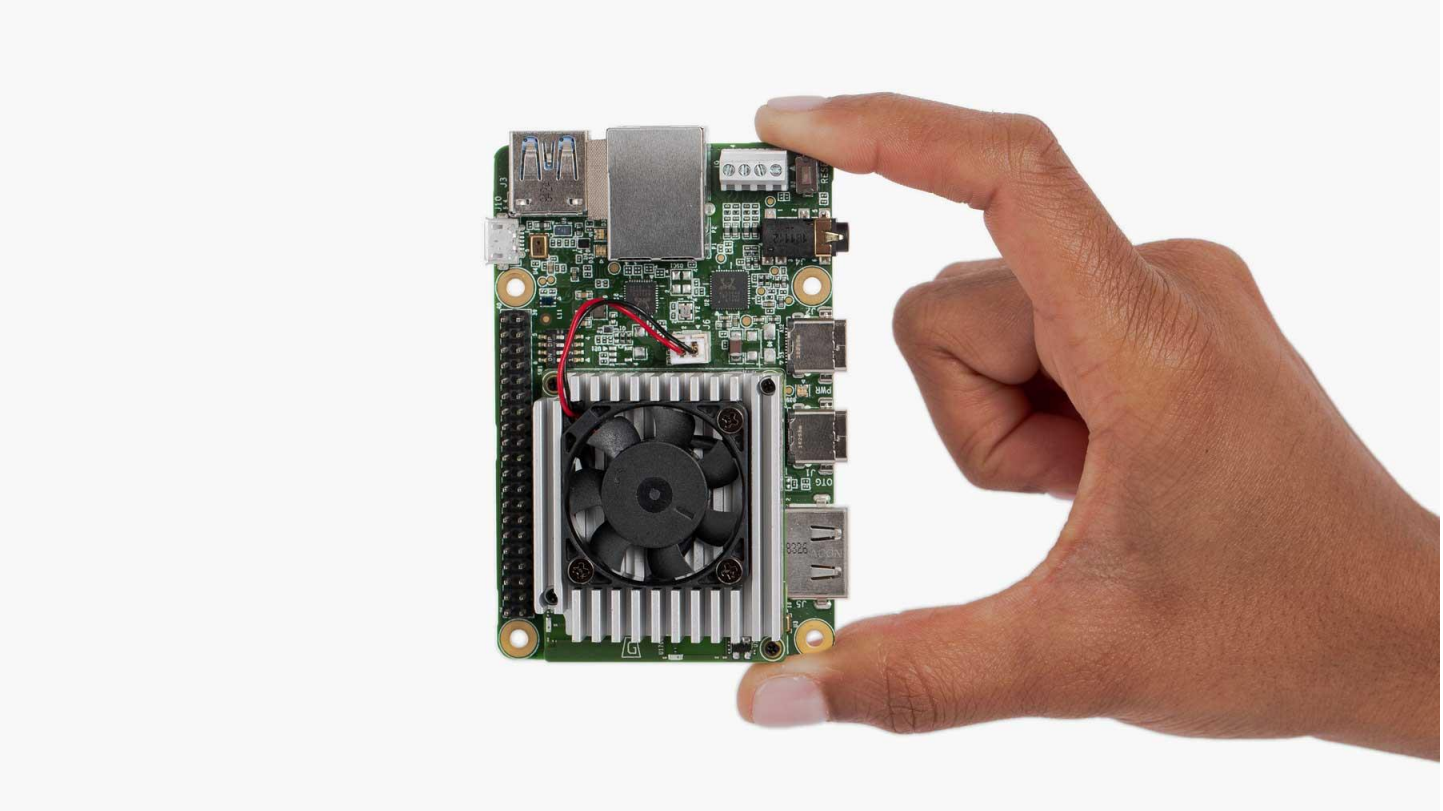
Get started with the Dev Board

The Coral Dev Board is a single-board computer that contains an Edge TPU coprocessor. It's ideal for prototyping new projects that demand fast on-device inferencing for machine learning models. This page is your guide to get started.
The setup requires flashing Mendel Linux to the board, and then accessing the board's shell terminal. Once you have terminal access and update some of the software, we'll show you how to run an image classification model on the board.
If you want to learn more about the hardware, see the Dev Board datasheet.



Warning: Use caution when handling the board to avoid electrostatic discharge or contact with conductive materials (metals). Failure to properly handle the board can result in a short circuit, electric shock, serious injury, death, fire, or damage to your board and other property.

1: Gather requirements

Note: Do not power the board or connect any cables until instructed to do so.

Before you begin, collect the following hardware:

- A host computer running Linux (recommended), Mac, or Windows 10
    - Python 3 installed
- One microSD card with at least 8 GB capacity, and an adapter to connect it to your host computer
- One USB-C power supply (2-3 A / 5 V), such as a phone charger
- One USB-C to USB-A cable (to connect to your computer)

- An available Wi-Fi connection (or Ethernet cable)

Note: Although you can connect a keyboard and monitor to the board, we do not recommend it because the system is not designed to operate as a desktop environment and doing so can affect the system performance. So our documentation emphasizes use of a terminal when interacting with the Dev Board (either with the serial console or SSH).

Other Windows requirements

For compatibility with our command-line instructions and the shell scripts in our example code, we recommend you use the Git Bash terminal on Windows, which is included with Git for Windows. You should install it now and open the Git Bash terminal for all the command-line instructions below.

However, as of Git for Windows v2.28, the Git Bash terminal cannot directly access Python. So after you install Git Bash, open it and run the following commands to make Python accessible:

```
echo  "alias python3='winpty python3.exe'" >> ~/.bash_profile

source ~/.bash_profile
```

2: Flash the board

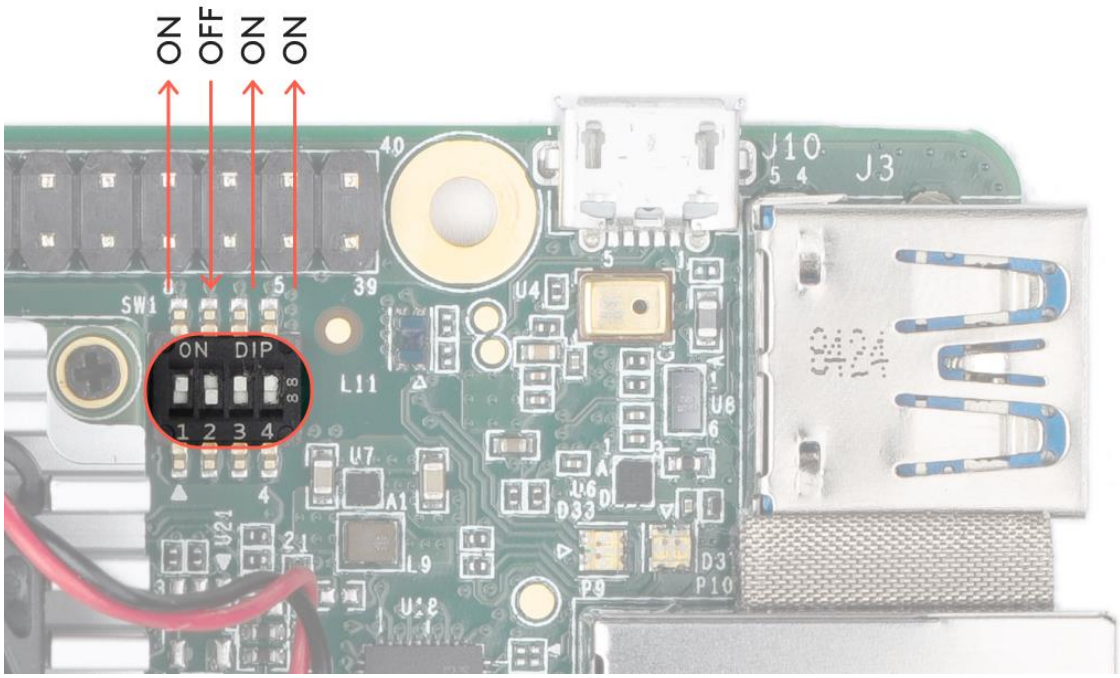
The Dev Board's factory settings do not include a system image (it includes only the U-Boot bootloader), so you need to flash the board with Mendel Linux.
To simplify the flashing process, we created a runtime system that runs on a microSD card and flashes the board with the Mendel system image. Unlike other boards like Raspberry Pi, Mendel Linux *does not* run on the microSD card—we only use the microSD card to install the operating system on the built-in flash memory—you'll remove the microSD card at the end of this process. (If you don't have a microSD card, you can instead manually flash the board over USB.)
Caution: This procedure is intended for new boards only. If you already installed Mendel, this **deletes all system and user data**—if you want to keep your user data, instead see the guide to flash a new system image.

To flash your board using the microSD card, follow these steps:

1. Download and unzip the SD card image: enterprise-eagle-flashcard-20211117215217.zip
   The ZIP contains one file named `flashcard_arm64.img`.
2. Use a program such as balenaEtcher to flash the `flashcard_arm64.img` file to your microSD card.

   This takes 5-10 minutes, depending on the speed of your microSD card and adapter.
3. While the card is being flashed, make sure your Dev Board is completely unplugged, and change the boot mode switches to boot from SD card, as shown in figure 1.

Boot mode	Switch 1	Switch 2	Switch 3	Switch 4
SD card	ON	OFF	ON	ON

4. 

5. **Figure 1.** Boot switches set to SD card mode
6. Once the card is flashed, safely remove it from your computer and insert it into the Dev Board (the card's pins face toward the board). The board should **not** be powered on yet.

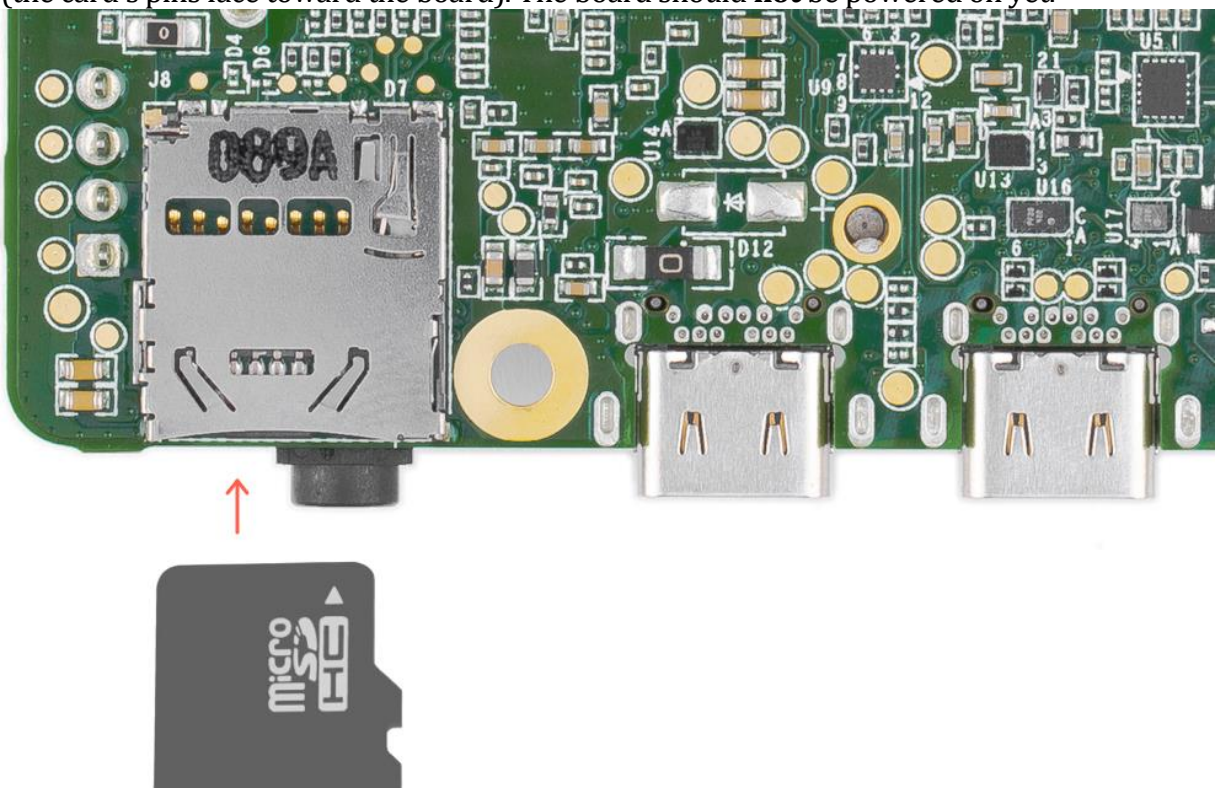


Figure 2. The microSD card slot is on the bottom

7. **Optional:** If you'd like to see the bootloader logs while the board is being flashed, either connect a monitor to the board's HDMI port or connect to the board's serial console. Nothing will appear until you power the board in the next step.
8. Power up the board by connecting your 2-3 A power cable to the USB-C port labeled "PWR" (see figure 3). The board's red LED should turn on.
   Caution: Do not attempt to power the board by connecting it to your computer.

When the board boots up, it loads the SD card image and begins flashing the board's eMMC memory.

It should finish in 5-10 minutes, depending on the speed of your microSD card.

You'll know it's done when the board shuts down and the red LED turns off.

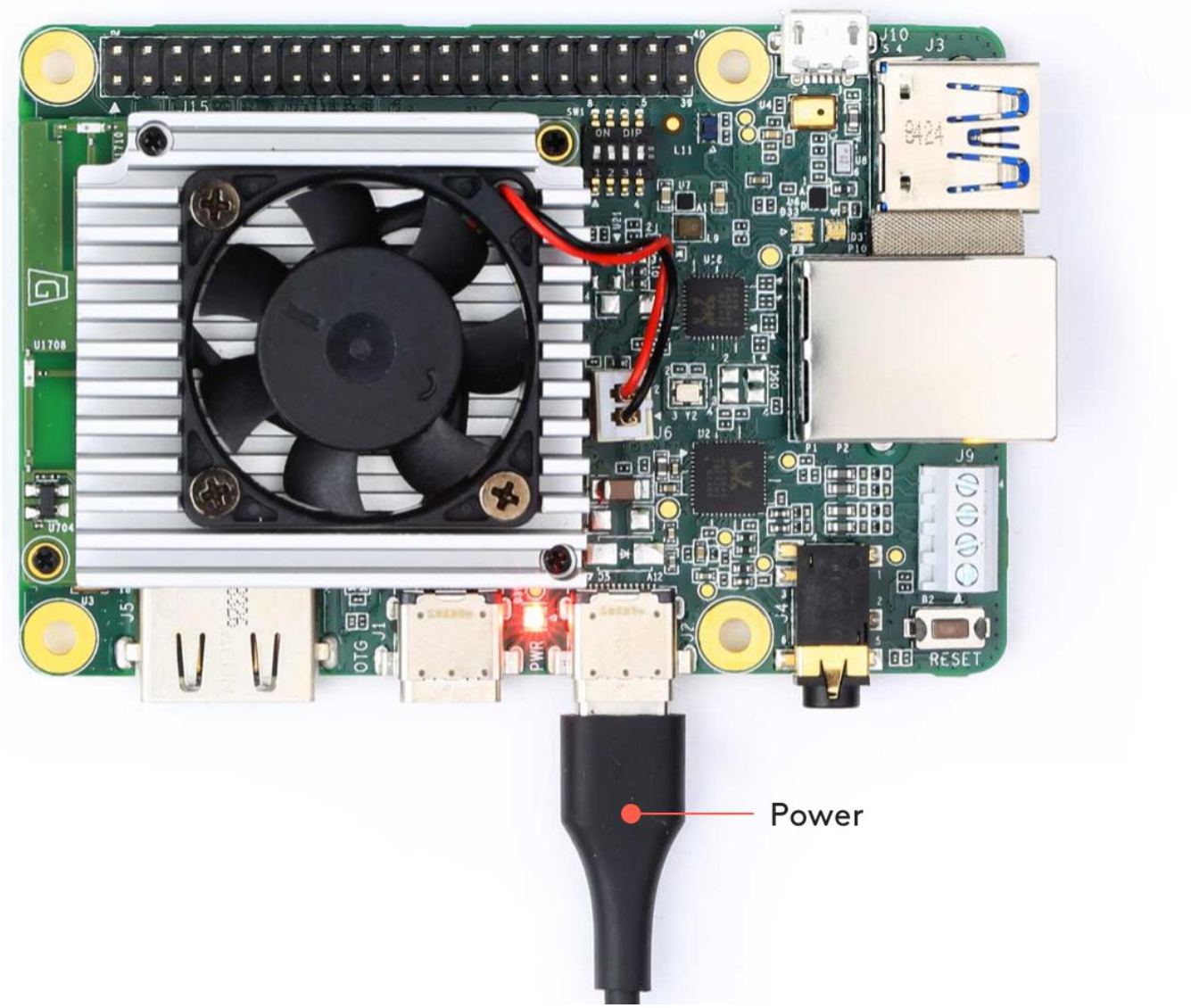


Figure 3. A USB-C power cable connected to the board

9. When the red LED turns off, unplug the power and remove the microSD card.
10. Change the boot mode switches to eMMC mode, as shown in figure 4:

Boot mode	Switch 1	Switch 2	Switch 3	Switch 4
eMMC	ON	OFF	OFF	OFF

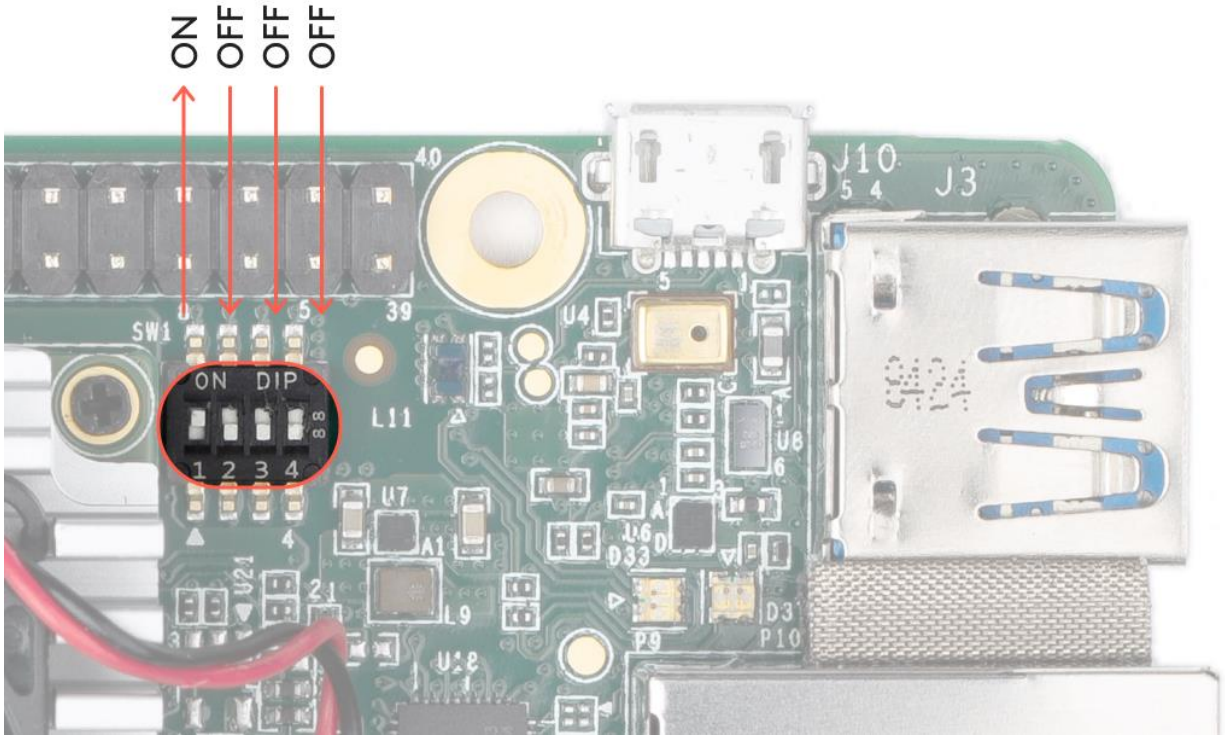
11. 

Figure 4. Boot switches set to eMMC mode

12. Connect the board to power and it should now boot up Mendel Linux.

    Booting up for the first time after flashing takes about 3 minutes (subsequent boot times are much faster).

3: Install MDT

While the board boots up, you can install MDT on your host computer.

MDT is a command line tool that helps you interact with the Dev Board. For example, MDT can list connected devices, install Debian packages on the board, and open a shell terminal on the board.

You can install MDT on your host computer follows:

```
python3 -m pip install --user mendel-development-tool
```

You might see a warning that `mdt` was installed somewhere that's not in your `PATH` environment variable. If so, be sure you add the given location to your `PATH`, as appropriate for your operating system. If you're on Linux, you can add it like this:

```
echo 'export PATH="$PATH:$HOME/.local/bin"' >> ~/.bash_profile

source ~/.bash_profile
```

Windows users: If you're using Git Bash, you'll need an alias to run MDT. Run this in your Git Bash terminal:

```
echo "alias mdt='winpty mdt'" >> ~/.bash_profile

source ~/.bash_profile
```

4: Connect to the board's shell via MDT

Mac users: Starting with macOS 10.15 (Catalina), you cannot create an MDT (or other SSH) connection over USB. However, MDT does work over your local network, so follow the instructions to use MDT on macOS (return here to complete the setup).
Now that you have the Mendel system on the board, you can initiate a secure shell session using the MDT tool. Using MDT is just an easy way to generate an OpenSSH public/private key pair, push the public key onto the board, and then establish an SSH connection. (You should have already installed MDT on your host computer, as per the above requirements.)

First, connect a USB-C cable from your computer to the board's other USB port (labeled "OTG").

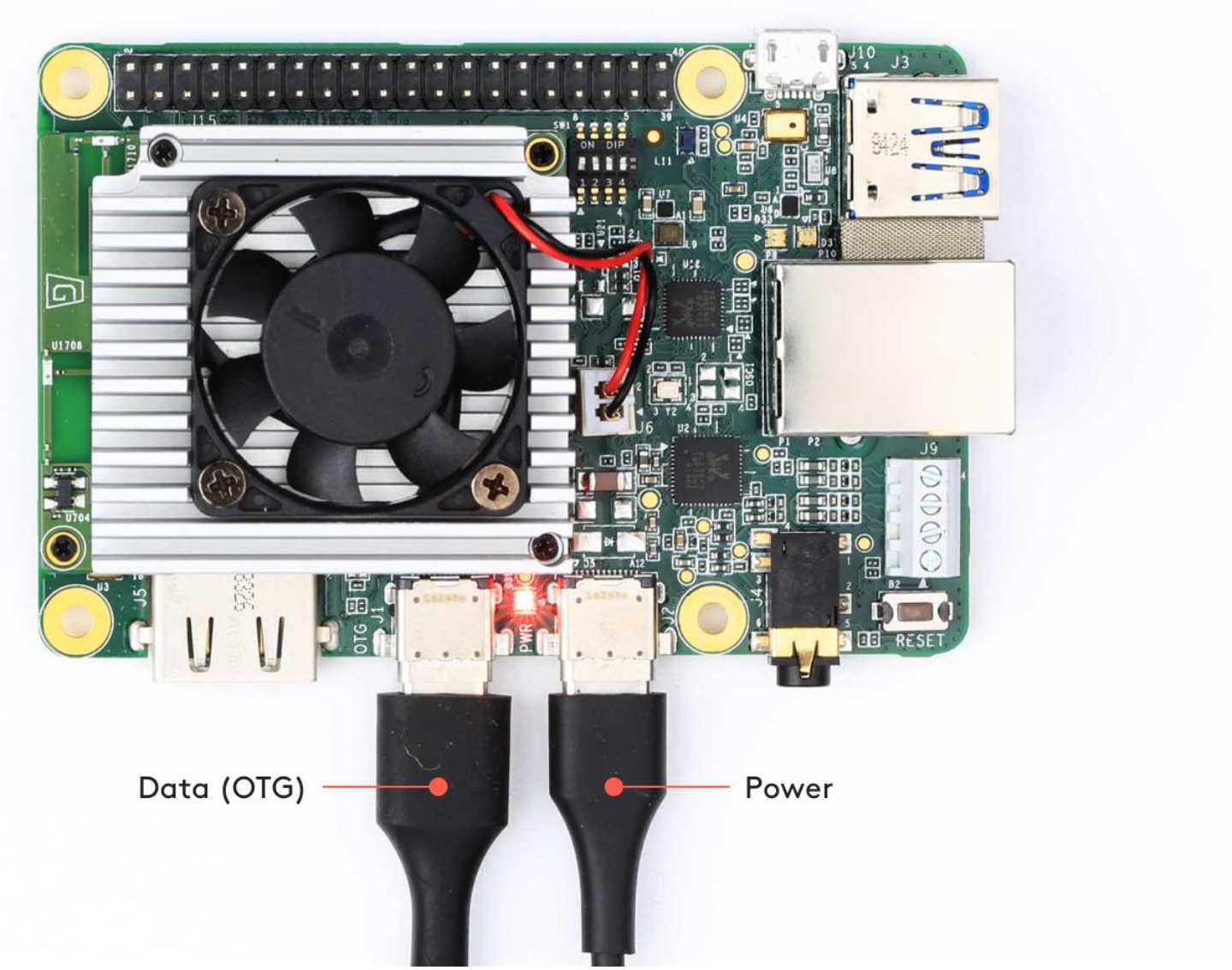


Figure 5. The USB data and power cables connected

Now make sure MDT can see your device by running this command from your host computer:

```
mdt devices
```

You should see output showing your board hostname and IP address:

```
orange-horse          (192.168.100.2)
```

If you don't see your device printed, it's probably because the system is still setting up Mendel. So instead run `mdt wait-for-device`. When your board is ready, it will print "Found 1 devices."
Note: Your board's hostname is randomly generated the first time it boots from a new flashing. We do this to ensure that each device within a local fleet is likely to have a unique name. Of course, you can change this name using standard Linux hostname tooling (such as `hostname`).

Now to open the device shell, run this command:

```
mdt shell
```

After a moment, you should see the board's shell prompt.
Note: You must connect to the board via MDT over USB at least once before you can SSH via any other method. Using USB allows MDT to generate an SSH public/private key pair and push it to the board's `authorized_keys` file, which then allows you to authenticate with SSH. (Using MDT is just easier than manually copying the key over the serial console.)
For more information about what you can do with MDT, read about the Mendel Development Tool.

5: Connect to the internet

You'll need the board online to download system updates, models, and samples.

Either connect an Ethernet cable to the board or select a Wi-Fi network by running the following command in the device shell:

```
nmtui
```

Then select `Activate a connection` and select a network from the list under `Wi-Fi (wlan0)`.

Alternatively, use the following command to connect to a known network name:

```
nmcli dev wifi connect <NETWORK_NAME> password <PASSWORD> ifname wlan0
```

Verify your connection with this command:

```
nmcli connection show
```

You should see your selected network listed in the output. For example:

```
NAME                UUID                                  TYPE             DEVICE
MyNetworkName       61f5d6b2-5f52-4256-83ae-7f148546575a  802-11-wireless  wlan0
```

The `DEVICE` name is `wlan0` for a Wi-Fi connection or `eth0` for an Ethernet connection.

6: Update the Mendel software

Some of our software updates are delivered with Debian packages separate from the system image, so make sure you have the latest software by running the following commands.

```
sudo apt-get update

sudo apt-get dist-upgrade
```

Now you're ready to run a TensorFlow Lite model on the Edge TPU!
Note: The `dist-upgrade` command updates all system packages for your current Mendel version. If you want to upgrade to a newer version of Mendel, you need to flash a new system image.

7: Run our demo app

For a video demo of the Edge TPU performance, run the following command from the Dev Board terminal:

```
edgetpu_demo --stream
```

Then on your desktop (that's connected to the Dev Board)—if you're connected to the board using MDT over USB—open 192.168.100.2:4664 in a browser. If you're instead connected to the board by other means (such as SSH over LAN or with an Ethernet cable), type the appropriate IP address into your browser with port 4664.

You should then see a video playing in your browser. The footage of the cars is a recording, but the MobileNet model is executing in real time on your Dev Board to detect each car.

Or if you have a monitor attached to the Dev Board, you can instead see the demo on that screen:

```
edgetpu_demo --device
```

Caution: Avoid touching the heat sink during operation. Whether or not the fan is running, the heat sink can become very hot to the touch and might cause burn injuries.

8: Run a model using the PyCoral API

The demo above is rather complicated, so we've created some simple examples that demonstrate how to perform an inference on the Edge TPU using the TensorFlow Lite API (assisted by the PyCoral API).

Execute the following commands from the Dev Board shell to run our image classification example:

1. Download the example code from GitHub:

```
mkdir coral && cd coral

git clone https://github.com/google-coral/pycoral.git

cd pycoral
```

6. Download the model, labels, and bird photo:

```
bash examples/install_requirements.sh classify_image.py
```

7. Run the image classifier with the bird photo (shown in figure 6):

```
python3 examples/classify_image.py \

--model test_data/mobilenet_v2_1.0_224_inat_bird_quant_edgetpu.tflite \

--labels test_data/inat_bird_labels.txt \
--input test_data/parrot.jpg
```

Figure 6. parrot.jpg

You should see results like this:

```
----INFERENCE TIME----
Note: The first inference on Edge TPU is slow because it includes loading the model into Edge TPU
memory.
13.1ms
2.7ms
3.1ms
3.2ms
3.1ms
-------RESULTS--------
Ara macao (Scarlet Macaw): 0.75781
```

Congrats! You just performed an inference on the Edge TPU using TensorFlow Lite.

To demonstrate varying inference speeds, the example repeats the same inference five times. It prints the time to perform each inference and then the top classification result (the label ID/name and the confidence score, from 0 to 1.0).
To learn more about how the code works, take a look at the `classify_image.py` source code and read about how to run inference on the Edge TPU with Python.
Note: The example above uses the TensorFlow Lite in Python, but you can also run an inference using C++. For information about each option, read the Edge TPU inferencing overview.

Next steps

If you got the Coral Camera, see the guide to connect a camera to the Dev Board.
To run some other types of neural networks, check out our example projects, including examples that perform real-time object detection, pose estimation, keyphrase detection, on-device transfer learning, and more.

If you want to train your own TensorFlow model for the Edge TPU, try these tutorials:

- Retrain an image classification model (MobileNet) (runs in Google Colab)
- Retrain an object detection model (EfficientDet) (runs in Google Colab)
- More model retraining tutorials are available on GitHub.
- Or to build your own model that's compatible with the Edge TPU, read TensorFlow Models on the Edge TPU

If you'd like to browse the Mendel source code and build it yourself for the Dev Board, take a look at the Mendel Getting Started guide.
Caution: When you're done with the Dev Board, **do not simply unplug the Dev Board**. Doing so could corrupt the system image if any write operations are in progress. Instead, safely shutdown the system with the following command:

```
sudo shutdown now
```

When the red LED on the Dev Board turns off, you can unplug the power.

Connect to the Dev Board I/O pins

The Dev Board provides access to several peripheral interfaces through the 40-pin expansion header, including GPIO, I2C, UART, and SPI. This page describes how you can interact with devices connected to these pins.
Because the Dev Board runs Mendel Linux, you can interact with the pins from user space using Linux interfaces such as device files (`/dev`) and sysfs files (`/sys`). There are also several API libraries you can use to program the peripherals connected to these pins. This page describes a few API options, including python-periphery, Adafruit Blinka, and libgpiod.

All I/O pins on the 40-pin header are powered by the 3.3 V power rail, with a programmable impedance of 40-255 ohms, and a max current of ~82 mA.

Figure 1. Default pin functions on the 40-pin header

Warning: When handling the I/O pins, be cautious to avoid electrostatic discharge or contact with conductive materials (metals). Failure to properly handle the board can result in a short circuit, electric shock, serious injury, death, fire, or damage to your board and other property.

Header pinout

Table 1 shows the header pinout, including the device or sysfs file for each pin, plus the character device numbers. For a pinout that includes the SoC pin names see the Dev Board datasheet instead. You can also see the header pinout from the command line by typing `pinout` on the Dev Board.

Note: All I/O pins have a 90k pull-down resistor inside the iMX8M SoC that is used by default during bootup, except for the I2C pins, which instead have a pull-up to 3.3 V on the SoM.

Caution: Do not connect a device that draws more than ~82 mA of power or you will brownout the system.

Table 1. Pinout for the Dev Board 40-pin header, with device file names and character device IDs (chip_number, line_number)

Chip, line	Device path	Pin function	Pin		Pin function	Device path	Chip, line
		+3.3 V	1	2	+5 V		
	/dev/i2c-1	I2C2_SDA	3	4	+5 V		
	/dev/i2c-1	I2C2_SCL	5	6	Ground		
	/dev/ttymxc2	UART3_TXD	7	8	UART1_TXD	/dev/ttymxc0	
		Ground	9	10	UART1_RXD	/dev/ttymxc0	
	/dev/ttymxc2	UART3_RXD	11	12	SAI1_TXC		
0, 6	/sys/class/gpio/gpio6	GPIO_P13	13	14	Ground		

Table 1. Pinout for the Dev Board 40-pin header, with device file names and character device IDs (chip_number, line_number)

Chip, line	Device path	Pin function	Pin		Pin function	Device path	Chip, line
2, 0	/sys/class/pwm/pwmchip2/pwm0	PWM3	15	16	GPIO_P16	/sys/class/gpio/gpio73	2, 9
		+3.3 V	17	18	GPIO_P18	/sys/class/gpio/gpio138	4, 10
	/dev/spidev0	ECSPI1_MOSI	19	20	Ground		
	/dev/spidev0	ECSPI1_MISO	21	22	GPIO_P22	/sys/class/gpio/gpio140	4, 12
	/dev/spidev0	ECSPI1_SCLK	23	24	ECSPI1_SS0	/dev/spidev0.0	
		Ground	25	26	ECSPI1_SS1	/dev/spidev0.1	
	/dev/i2c-2	I2C3_SDA	27	28	I2C3_SCL	/dev/i2c-2	
0, 7	/sys/class/gpio/gpio7	GPIO_P29	29	30	Ground		
0, 8	/sys/class/gpio/gpio8	GPIO_P31	31	32	PWM1	/sys/class/pwm/pwmchip0/pwm0	0, 0
1, 0	/sys/class/pwm/pwmchip1/pwm0	PWM2	33	34	Ground		
		SSAI1_TXFS	35	36	GPIO_P36	/sys/class/gpio/gpio141	4, 13
2, 13	/sys/class/gpio/gpio77	GPIO_P37	37	38	SAI1_RXD0		
		Ground	39	40	SAI1_TXD0		

For further information on the various interfaces, see the i.MX 8M Dual/8M QuadLite/8M Quad Applications Processors Reference Manual.

Program with python-periphery

The python-periphery library provides a generic Linux interface that's built atop the sysfs and character device interface, providing APIs to control GPIO, PWM, I2C, SPI, and UART pins.

You can install the library on your Dev Board as follows:

```
python3 -m pip install python-periphery
```

The Periphery library allows you to select a GPIO or PWM pin with a pin number. Other interfaces, such as I2C and UART pins must be specified using the pin's device path. See the following examples.
Note: The Synchronous Audio Interface (SAI) pins are not accessible using python-periphery. For details, see the i.MX 8M reference manual.

GPIO

You can instantiate a GPIO object using either the sysfs path (deprecated) or the character device numbers.

The following code instantiates each GPIO pin as input using the character devices:

```
gpio_p13 = GPIO("/dev/gpiochip0", 6, "in")
gpio_p18 = GPIO("/dev/gpiochip4", 10, "in")
gpio_p22 = GPIO("/dev/gpiochip4", 12, "in")
gpio_p29 = GPIO("/dev/gpiochip0", 7, "in")
gpio_p31 = GPIO("/dev/gpiochip0", 8, "in")
gpio_p36 = GPIO("/dev/gpiochip4", 13, "in")

gpio_p16 = GPIO("/dev/gpiochip2", 9, "out")
gpio_p37 = GPIO("/dev/gpiochip2", 13, "out")
```

Note: GPIO_P16 and GPIO_P37 currently support only the "out" direction.

For example, here's how to turn on an LED when you push a button:

```
from periphery import GPIO

led = GPIO("/dev/gpiochip2", 13, "out")   # pin 37
button = GPIO("/dev/gpiochip4", 13, "in")   # pin 36

try:
  while True:
    led.write(button.read())
finally:
  led.write(False)
  led.close()
  button.close()
```

For more examples, see the periphery GPIO documentation.

PWM

The following code shows how to instantiate each of the PWM pins with Periphery:

```
pwm1 = PWM(0, 0)
pwm2 = PWM(1, 0)
pwm3 = PWM(2, 0)
```

For usage examples, see the periphery PWM documentation.

I2C

The following code shows how to instantiate each of the I2C ports with Periphery:

```
i2c2 = I2C("/dev/i2c-1")
i2c3 = I2C("/dev/i2c-2")
```

For usage examples, see the periphery I2C documentation.

SPI

The following code shows how to instantiate each of the SPI ports with Periphery:

```
# SPI1, SS0, Mode 0, 10MHz
```

```
spi1_0 = SPI("/dev/spidev0.0", 0, 10000000)
# SPI1, SS1, Mode 0, 10MHz
spi1_1 = SPI("/dev/spidev0.1", 0, 10000000)
```

For usage examples, see the periphery SPI documentation.

Help! If you receive a `Permission denied` error when trying to access the SPI device, it should be fixed if you run the following:

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

sudo reboot now
```

UART

The following code shows how to instantiate each of the UART ports with Periphery:

```
# UART1, 115200 baud
uart1 = Serial("/dev/ttymxc0", 115200)
# UART3, 9600 baud
uart3 = Serial("/dev/ttymxc2", 9600)
```

Caution: UART1 is shared with the Linux serial console. To use the UART1 port in your application, you must disable the serial console with the following command:

```
systemctl stop serial-getty@ttymxc0.service
```

For usage examples, see the periphery Serial documentation.

Program with Adafruit Blinka

The Blinka library not only offers a simple API for GPIO, PWM, I2C, and SPI, but also provides compatibility with a long list of sensor libraries built for CircuitPython. That means you can reuse CircuitPython code for peripherals that was originally used on microcontrollers or other boards such as Raspberry Pi.

To get started, install Blinka and libgpiod on your Dev Board Mini as follows:

```
sudo apt-get install python3-libgpiod

python3 -m pip install adafruit-blinka
```

Then you can turn on an LED when you push a button as follows (notice this uses pin names from the pinout above):

```
import board
import digitalio

led = digitalio.DigitalInOut(board.GPIO_P37)  # pin 37
led.direction = digitalio.Direction.OUTPUT

button = digitalio.DigitalInOut(board.GPIO_P36)  # pin 36
button.direction = digitalio.Direction.INPUT

try:
  while True:
    led.value = button.value
finally:
  led.value = False
  led.deinit()
  button.deinit()
```

For more information, including example code using I2C and SPI, see the Adafruit guide for CircuitPython libraries on Coral. But we suggest you skip their setup guide and install Blinka as shown above. Also check out the Blinka API reference.

Program GPIOs with libgpiod

You can also interact with the GPIO pins using the libgpiod library, which provides both C++ and Python API bindings. But libgpiod is for GPIOs only, not any digital protocols. (The Blinka library uses libgpiod as its implementation for GPIOs.)

There's currently no online API docs for libgpiod, but the source code is fully documented. If you clone the repo, you can build C++ docs with Doxygen. For Python, you can install the libgpiod package and print the API docs as follows:

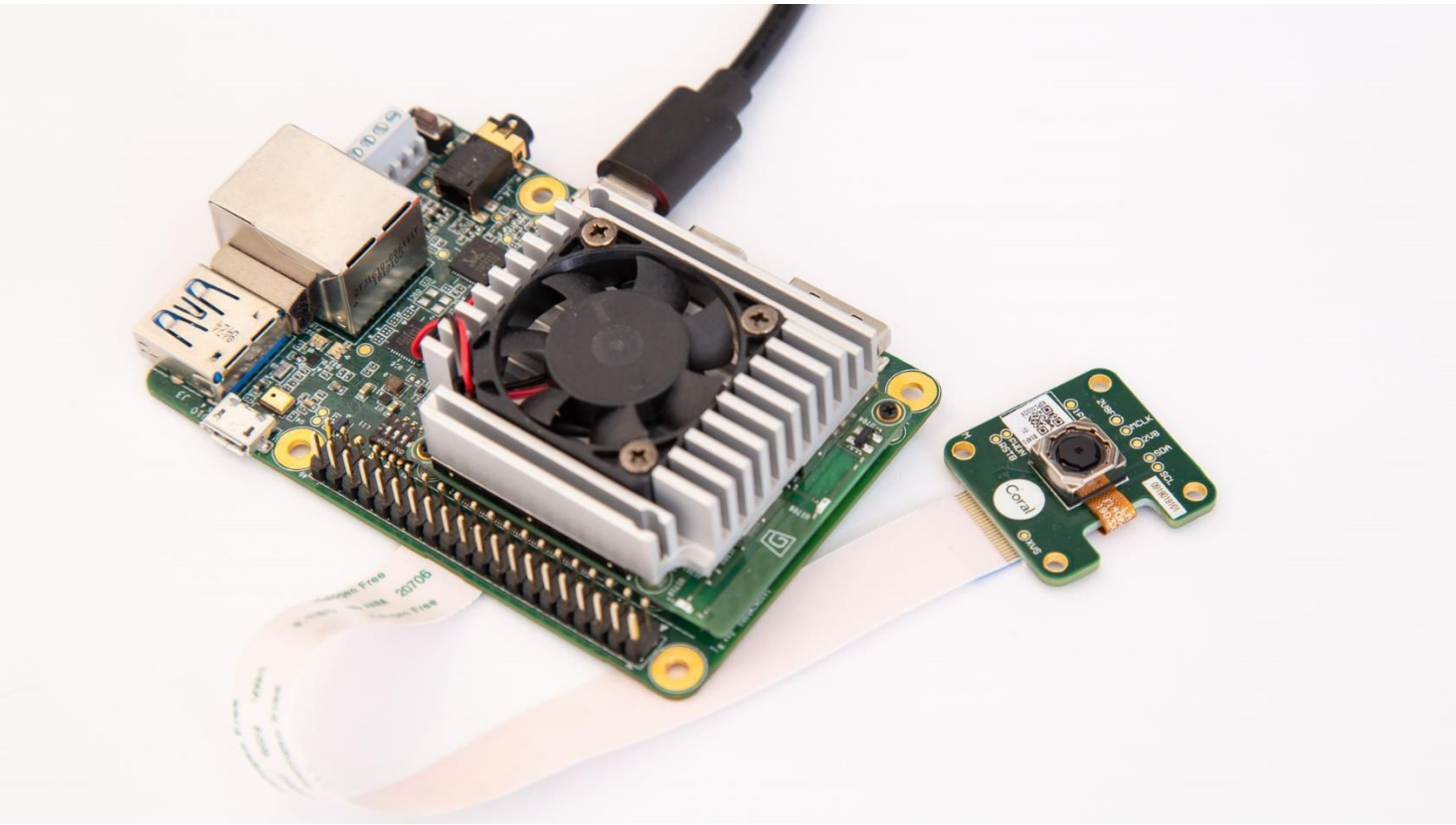
```
sudo apt-get install python3-libgpiod

python3 -c 'import gpiod; help(gpiod)'
```

Then you can turn on an LED when you push a button as follows:

```
import gpiod

CONSUMER = "led-demo"
chip2 = gpiod.Chip("2", gpiod.Chip.OPEN_BY_NUMBER)
chip4 = gpiod.Chip("4", gpiod.Chip.OPEN_BY_NUMBER)

led = chip2.get_line(13)   # pin 37
led.request(consumer=CONSUMER, type=gpiod.LINE_REQ_DIR_OUT, default_vals=[0])
button = chip4.get_line(13)   # pin 36
button.request(consumer=CONSUMER, type=gpiod.LINE_REQ_DIR_IN)

try:
  while True:
    led.set_value(button.get_value())
finally:
  led.set_value(0)
  led.release()
  button.release()
```

Connect the Coral Camera

The Coral Camera connects to the CSI connector on the bottom of the Dev Board.



You can connect the camera to the Dev Board as follows:

1. Make sure the board is powered off and unplugged.
2. On the bottom of the Dev Board, locate the CSI "Camera Connector" and flip the small black latch so it's facing upward, as shown in figure 1.

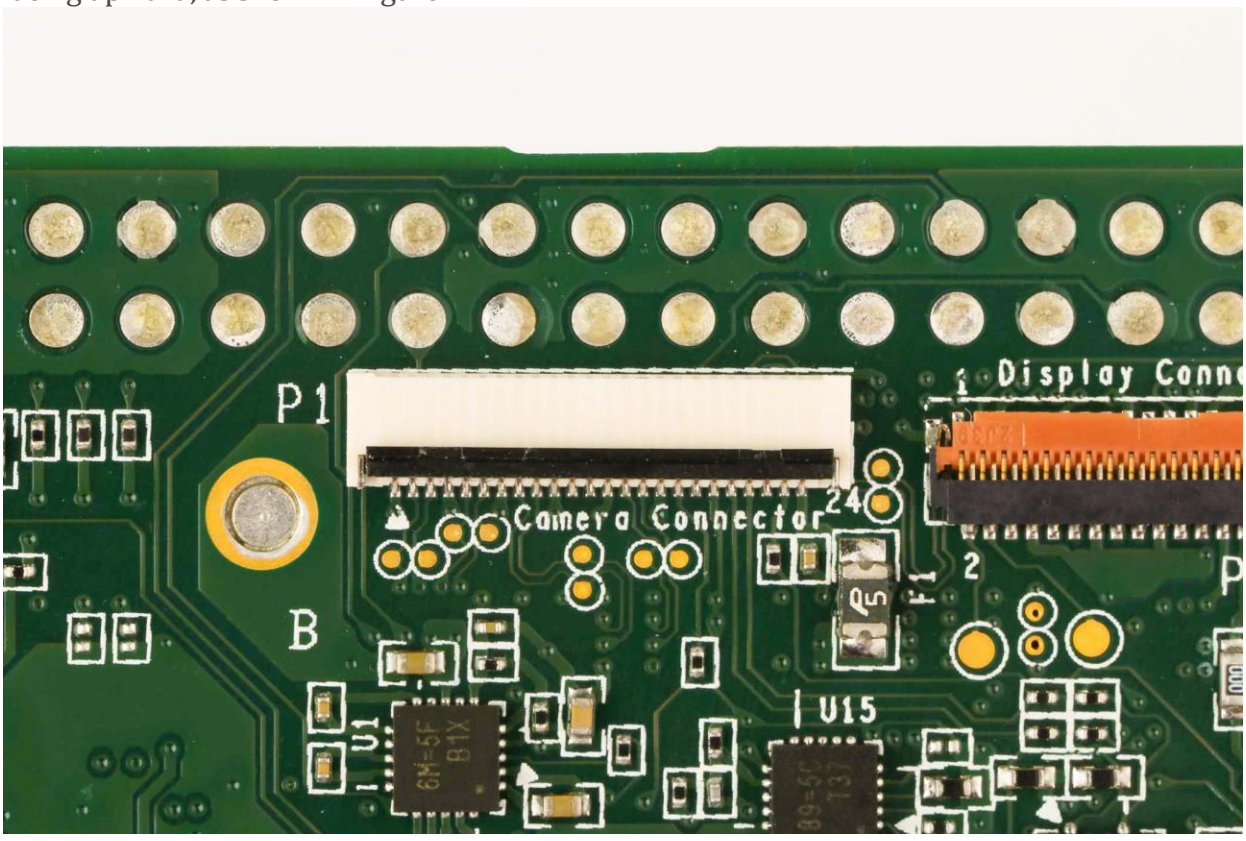


Figure 1. The camera connector with the latch open

3. Slide the flex cable into the connector with the contact pins facing toward the board (the blue strip is facing away from the board), as shown in figure 2.
4. Close the black latch.

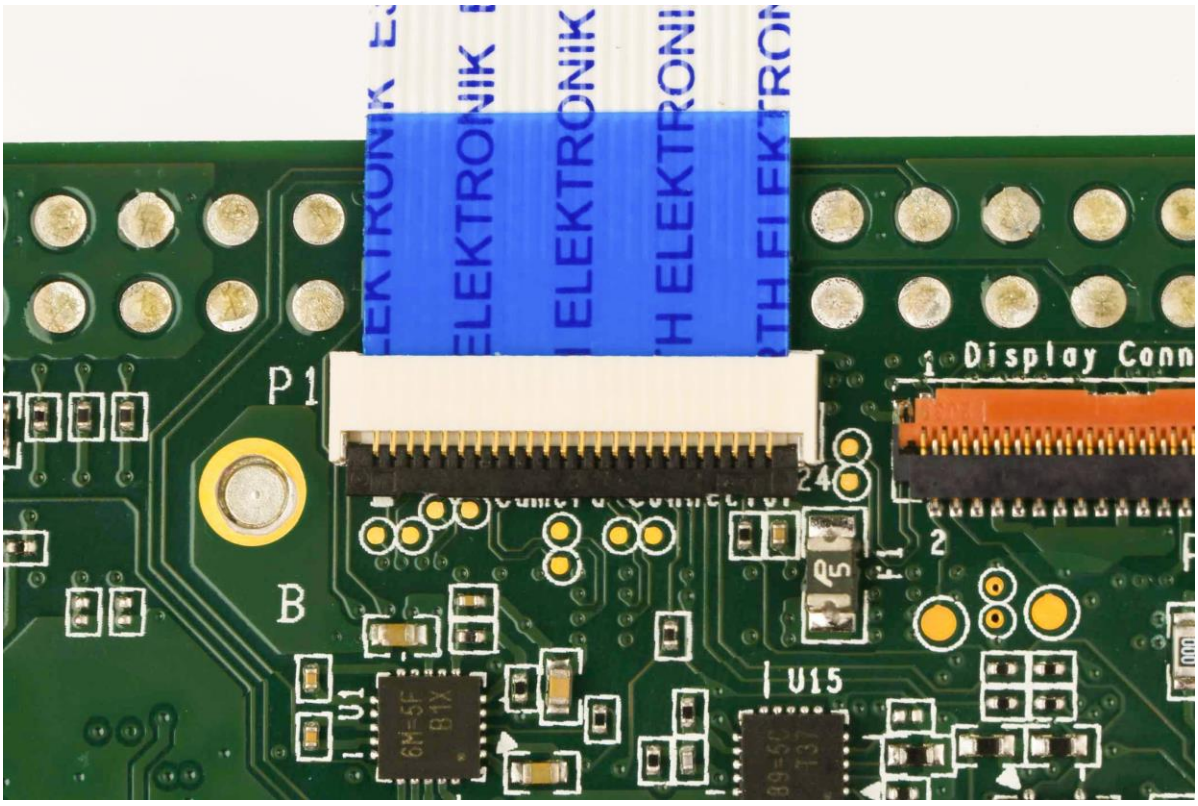


Figure 2. The camera cable inserted and the latch closed

5. Likewise, connect the other end of the flex cable to the matching connector on the camera module.

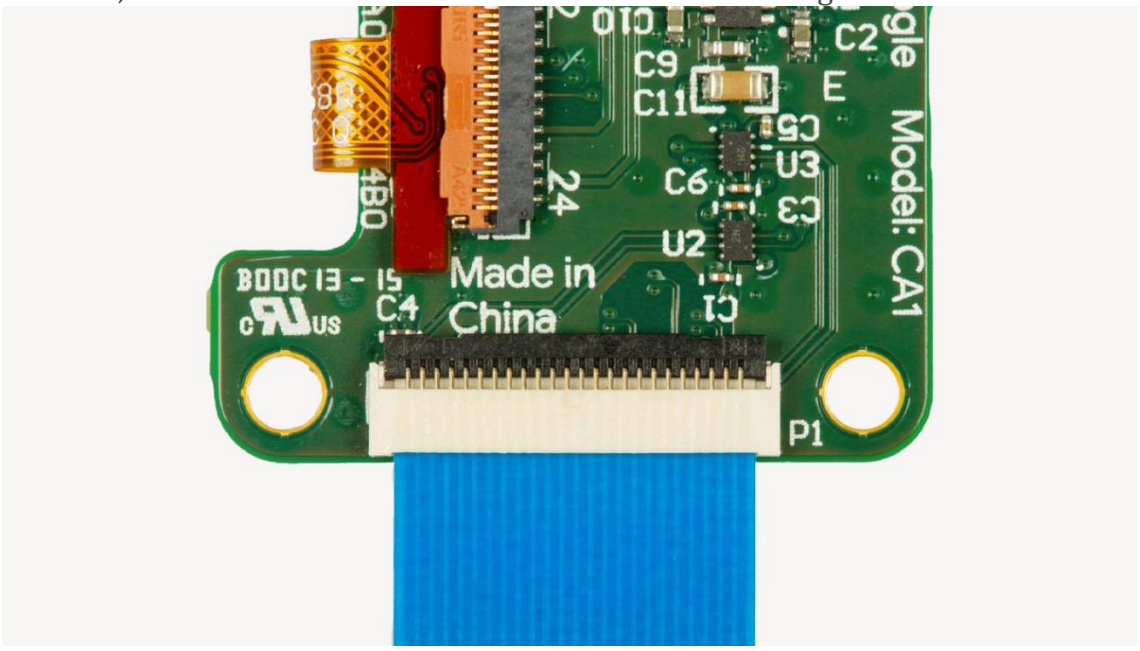


Figure 3. The camera cable inserted in the camera module

6. Power on the board and verify it detects the camera by running this command:

```
v4l2-ctl --list-devices
```

You should see the camera listed as `/dev/video0`:

```
i.MX6S_CSI (platform:30a90000.csi1_bridge):
```

```
        /dev/video0
```

For a quick camera test, connect to the board's shell terminal and run the snapshot tool:

```
snapshot
```

If you have a monitor attached to the board, you'll see the live camera feed.

You can press Spacebar to save an image to the home directory.
Then try running a model with the demo scripts below.

Connect a USB camera

Any USB camera that matches the USB UVC standard should be compatible with the Dev Board.

Just plug in the camera to the USB-A port. (It's okay if the board is already powered on.)

Then enter the following command to list the camera's supported video formats:

```
v4l2-ctl --list-formats-ext --device /dev/video1
```

You should see a long list of results that looks something like this:

```
ioctl: VIDIOC_ENUM_FMT
    Index       : 0
    Type        : Video Capture
    Pixel Format: 'YUYV'
    Name        : YUYV 4:2:2
        Size: Discrete 640x480
            Interval: Discrete 0.033s (30.000 fps)
            Interval: Discrete 0.042s (24.000 fps)
            Interval: Discrete 0.050s (20.000 fps)
            Interval: Discrete 0.067s (15.000 fps)
            Interval: Discrete 0.100s (10.000 fps)
            Interval: Discrete 0.133s (7.500 fps)
            Interval: Discrete 0.200s (5.000 fps)
```

Take note of the size and available FPS values. You'll need to pass those in the demo scripts below, though the default values shown below should work for most cameras.
Note: Be sure that your list includes `Pixel Format: 'YUYV'`. Currently, YUYV is the only format supported. But the commands below refer to this format with the name YUY2, which is just a different name for the same thing.

Run a demo with the camera

The Mendel system image on the Dev Board includes two demos that perform real-time image classification and object detection.

First, make sure you have the latest software on your board:

```
sudo apt-get update

sudo apt-get dist-upgrade
```

Note: The following demo code is optimized for performance on the Dev Board, and you can get the source code from the edgetpuvision Git repo. If you'd like to see other examples using a camera, which are more broadly applicable for other devices (not just the Dev Board), see the examples-camera GitHub repo.

Download the model files

Before you run either demo, you'll need to download the model files on your Dev Board.

First, set this environment variable:

```
export DEMO_FILES="$HOME/demo_files"
```

Then download the following models (be sure you're connected to the internet):

```
# The image classification model and labels file
wget -P ${DEMO_FILES}/ https://github.com/google-
coral/test_data/raw/master/mobilenet_v2_1.0_224_quant_edgetpu.tflite

wget -P ${DEMO_FILES}/ https://raw.githubusercontent.com/google-coral/test_data/release-
frogfish/imagenet_labels.txt

# The face detection model (does not require a labels file)
wget -P ${DEMO_FILES}/ https://github.com/google-
coral/test_data/raw/master/ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tflite
```

To run each demo, we've provided two ways to see the video and inference results:

- Using a monitor attached to the Dev Board via HDMI
- Using a streaming server that allows you to see the video from another computer's web browser when on the same network

View on a monitor

The following demos require that you have a monitor connected to the HDMI port on the Dev Board so you can see the video.
Note: By default, the Dev Board is locked at a 1920x1080 output, so your monitor must support this resolution or nothing will appear. If your monitor does not support 1920x1080, you can change the default video output.

Run the image classification model with a monitor

This demo classifies 1,000 different objects shown to the camera.

If you're using the Coral Camera:

```
edgetpu_classify \
--model ${DEMO_FILES}/mobilenet_v2_1.0_224_quant_edgetpu.tflite \
--labels ${DEMO_FILES}/imagenet_labels.txt
```

If you're using a USB camera:

```
edgetpu_classify \
--source /dev/video1:YUY2:800x600:24/1 \
--model ${DEMO_FILES}/mobilenet_v2_1.0_224_quant_edgetpu.tflite \
--labels ${DEMO_FILES}/imagenet_labels.txt
```

In the `--source` argument (for the USB camera only), you must specify 4 parameters using values printed during the USB camera setup:

- `/dev/video1` is the device file. Yours should be the same if it's the only attached camera.
- `YUY2` is the only supported pixel format (same as `YUYV`).
- `800x600` is the image resolution. This must match one of the resolutions listed for your camera.
- `24/1` is the framerate. It must also match one of the listed FPS values for the given format.

Run the face detection model with a monitor

This demo draws a box around any detected human faces.

If you're using the Coral Camera:

```
edgetpu_detect \
--model ${DEMO_FILES}/ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tflite
```

If you're using a USB camera:

```
edgetpu_detect \
--source /dev/video1:YUY2:800x600:24/1  \
--model ${DEMO_FILES}/ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tflite
```

See the previous section for details about the `--source` arguments.

View with a streaming server

These demos require that your Dev Board be network-accessible from another computer (such as when connected to the board shell via MDT) so you can see the camera output in a web browser.
Note: We recommend using Chrome to view the camera streams. Other browsers might not show the image overlays.

Run the image classification model with a streaming server

This demo classifies 1,000 different objects shown to the camera.

If you're using the Coral Camera:

```
edgetpu_classify_server \
--model ${DEMO_FILES}/mobilenet_v2_1.0_224_quant_edgetpu.tflite \
--labels ${DEMO_FILES}/imagenet_labels.txt
```

If you're using a USB camera:

```
edgetpu_classify_server \
--source /dev/video1:YUY2:800x600:24/1  \
--model ${DEMO_FILES}/mobilenet_v2_1.0_224_quant_edgetpu.tflite \
--labels ${DEMO_FILES}/imagenet_labels.txt
```

In the `--source` argument (for the USB camera only), you must specify 4 parameters using values printed during the USB camera setup:

- `/dev/video1` is the device file. Yours should be the same if it's the only attached camera.
- `YUY2` is the only supported pixel format (same as `YUYV`).
- `800x600` is the image resolution. This must match one of the resolutions listed for your camera.
- `24/1` is the framerate. It must also match one of the listed FPS values for the given format.

With either camera type, you should see the following message:

```
INFO:edgetpuvision.streaming.server:Listening on ports tcp: 4665, web: 4664, annexb: 4666
```

Which means your Dev Board is now hosting a streaming server. So from any computer that can access the board, you can view the camera stream at `http://<board_ip_address>:4664/`. For example, if you're connected to the board shell over USB, then go to http://192.168.100.2:4664/.

Run the face detection model with a streaming server

This demo draws a box around any detected human faces.

If you're using the Coral Camera:

```
edgetpu_detect_server \
--model ${DEMO_FILES}/ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tflite
```

If you're using a USB camera:

```
edgetpu_detect_server \
```

```
--source /dev/video1:YUY2:800x600:24/1  \
--model ${DEMO_FILES}/ssd_mobilenet_v2_face_quant_postprocess_edgetpu.tflite
```

See the previous section for details about the `--source` arguments.

With either camera type, you should see the following message:

```
INFO:edgetpuvision.streaming.server:Listening on ports tcp: 4665, web: 4664, annexb: 4666
```

Which means your Dev Board is now hosting a streaming server. So from any computer that can access the board, you can view the camera stream at `http://<board_ip_address>:4664/`. For example, if you're connected to the board shell via MDT, then go to http://192.168.100.2:4664/.

Try other example code

We have several other examples that are compatible with almost any camera and any Coral device with an Edge TPU (including the Dev Board). They each show how to stream images from a camera and run classification or detection models . Each example uses a different camera library, such as GStreamer, OpenCV, and PyGame.
To explore the code and run them, see the instructions at github.com/google-coral/examples-camera.

serial console

Although we recommend that you access the shell terminal with MDT, some situations require direct connection with the serial console. So this page shows you how to do that with the Dev Board.

You'll need the following items:

- Coral Dev Board
- Linux, Windows 10, or macOS computer
- USB-A to USB-micro-B cable (must be a USB data cable)
- 2 - 3 A (5 V) USB-C power supply

Note: If you're on Linux or Mac and your board is fully booted, you can instead open the serial console using the USB OTG port.

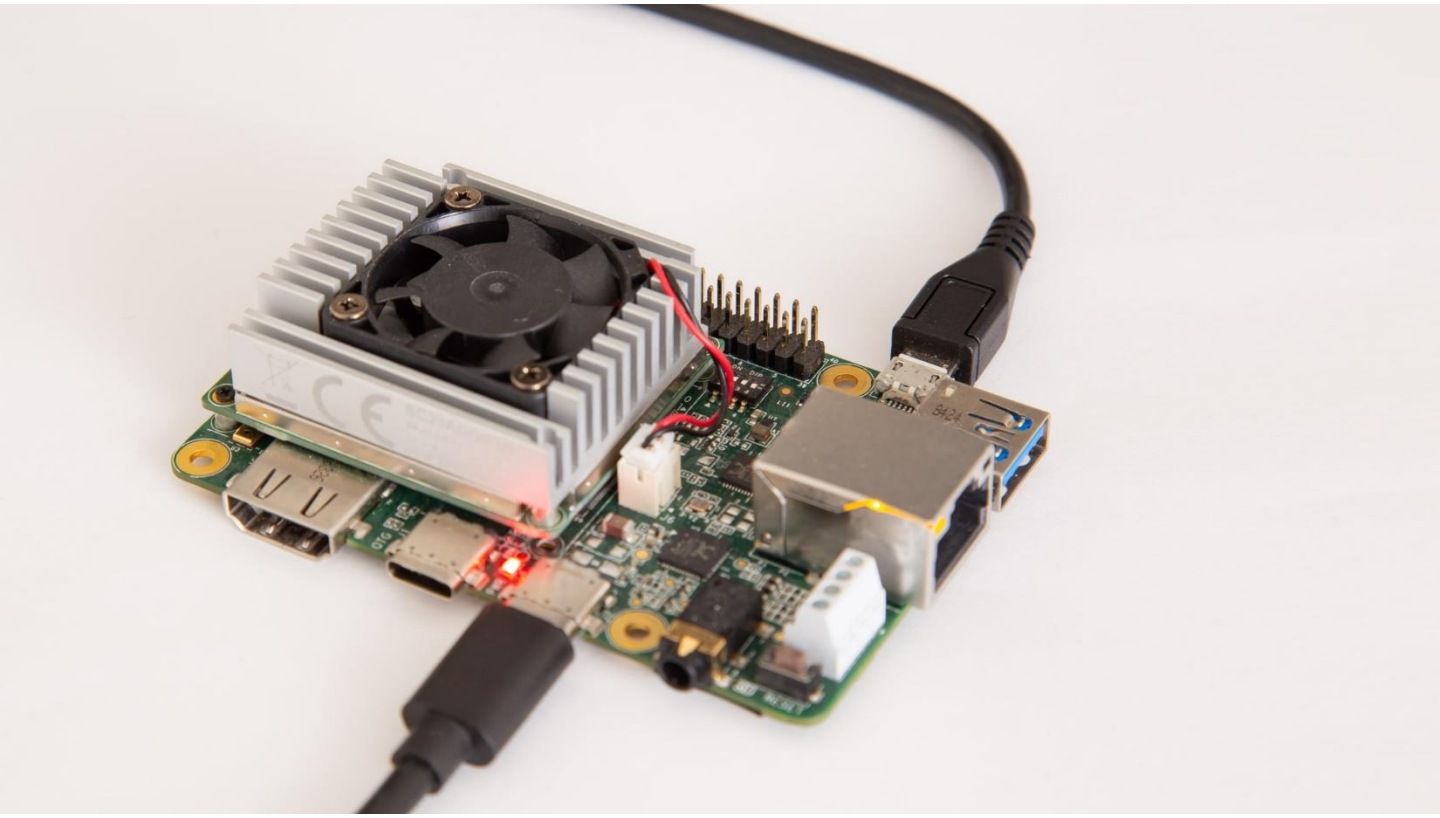


Figure 1. A micro-B USB cable connected to the Dev Board's serial console port

Connect with Linux

You can connect to the Dev Board's serial console from Linux as follows:

1. First make sure your Linux user account is in the `plugdev` and `dialout` system groups by running this command:

   ```
   sudo usermod -aG plugdev,dialout <username>
   ```

   Then reboot your computer for the new groups to take effect.

2. Connect your computer to the board with the micro-B USB cable, and connect the board to power, as shown in figure 1.

3. Determine the device filename for the serial connection by running this command on your Linux computer:

   ```
   dmesg | grep ttyUSB
   ```

   You should see two results such as this:

   ```
   [ 6437.706335] usb 2-13.1: cp210x converter now attached to ttyUSB0
   [ 6437.708049] usb 2-13.1: cp210x converter now attached to ttyUSB1
   ```

   If you don't see results like this, double-check your USB cable.

5. Then connect with this command (using the name of the first device listed as "cp210x converter"):

   ```
   screen /dev/ttyUSB0 115200
   ```

   Help! If it prints `Cannot access line '/dev/ttyUSB0'`, then your Linux user account is not in the `plugdev` and/or `dialout` system group. Ask your system admin to add your account to both groups, and then restart your computer for it to take effect.

If you see `[screen is terminating]`, it might also be due to the system groups, or there's something else wrong with `screen`—ensure all `screen` sessions are closed (type `screen -ls` to see open sessions), unplug the USB cable from the Dev Board, and then try again.

6. When the screen terminal opens, it will probably be blank. Hit Enter and you should see the login prompt once the system finishes booting up.

   The default username and password are both "mendel".

When you're done, exit the `screen` session by pressing Control+A, D.

Connect with Windows

You can connect to the Dev Board's serial console from Windows 10 as follows:

1. Connect your computer to the board with the micro-B USB cable, and connect the board to power, as shown in figure 1.
2. On your Windows computer, open **Device Manager** and find the board's COM port.
   Within a minute of connecting the USB cable, Windows should automatically install the necessary driver. So if you expand **Ports (COM & LPT)**, you should see two devices with the name "Silicon Labs Dual CP2105 USB to UART Bridge".

   Take note of the COM port for the device named "Enhanced COM Port" (such as "COM3"). You'll use it in the next step.
   If Windows cannot identify the device, it should instead be listed under **Other devices**. Right-click on **Enhanced Com Port** and select **Update driver** to find the appropriate device driver.
3. Open PuTTY or another serial console app and start a serial console connection with the above COM port, using a baud rate of `115200`. For example, if using PuTTY:
   1. Select **Session** in the left pane.
   2. For the **Connection type**, select **Serial**.
   3. Enter the COM port ("COM3") for **Serial line**, and enter "115200" for **Speed**.
   4. Then click **Open**.
4. When the screen terminal opens, it will probably be blank. Hit Enter and you should see the login prompt once the system finishes booting up.

   The default username and password are both "mendel".

Connect with macOS

You can connect to the Dev Board's serial console from macOS as follows:

1. Install the CP210x USB to UART Bridge VCP driver:
   Caution: Before installing the following package, be sure you've applied all available macOS software updates. Otherwise, you might be blocked from installing due to system security that disables the **Allow** button in System Preferences.
   1. Download the CP210x driver for macOS (see here for details).
   2. Unzip the package and install the driver.
2. Connect your computer to the board with the micro-B USB cable, and connect the board to power, as shown in figure 1.
3. Verify the CP210x driver is working by running this command:

```
ls /dev/cu*
```

You should see `/dev/cu.SLAB_USBtoUART` listed. If not, either there's a problem with your USB cable or the driver is not loaded. You can load the driver with `sudo kextload /Library/Extensions/SiLabsUSBDriver.kext` and then go to the system **Security & Privacy** preferences and click **Allow**. Also try unplugging the micro-USB cable on the Dev Board, then replug it and try again. Or, you also might need reboot your computer.

4. Then connect with this command:
```
screen /dev/cu.SLAB_USBtoUART 115200
```
5. When the screen terminal opens, it will probably be blank. Hit Enter and you should see the login prompt once the system finishes booting up.

   The default username and password are both "mendel".

When you're done, exit the `screen` session by pressing Control+A, D.

Connect over USB OTG

You can also connect to the serial console without the micro-B USB cable, but only if your host computer is Linux or Mac, and your board is fully booted up.
Note: The USB OTG port provides serial console access only when Mendel is booted up. As such, this approach does not provide access to the kernel boot logs or the u-boot command prompt. If you want those things or your board fails to boot Mendel, you must use the micro-B USB cable as described above.

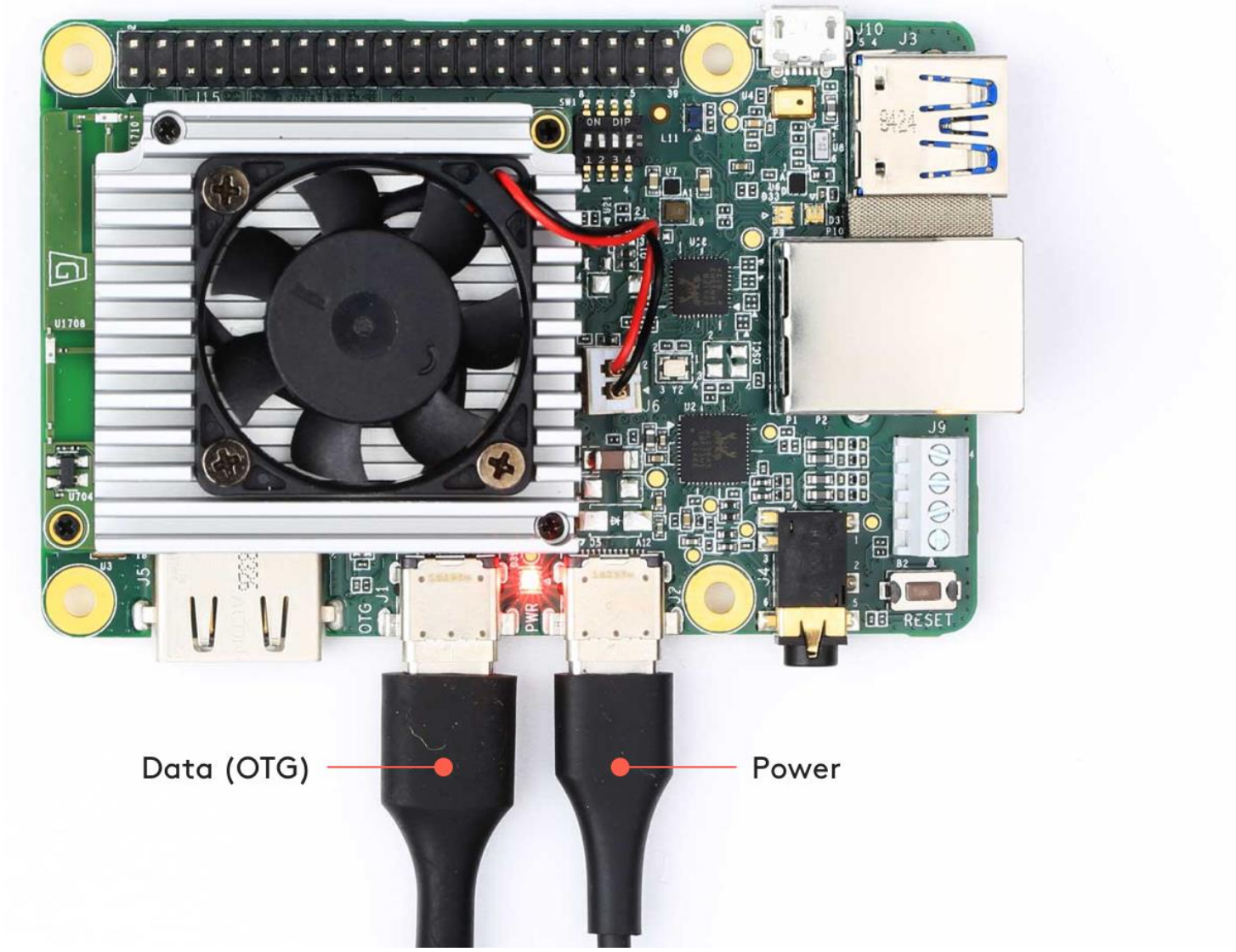


Figure 2. The USB OTG line can open the serial console

To connect to the serial console using the USB OTG port, follow these steps:

1. Make sure your board is fully booted up.
2. Connect a USB cable from your computer to the board's OTG port, and then run the following command in a terminal:
    - On Linux:
      ```
      ls /dev/ttyACM*
      ```
    - On Mac:
      ```
      ls /dev/cu.usbmodem*
      ```
3. Connect to the device shown using a serial console program such as `screen` as follows:
   ```
   screen /dev/ttyACM0 115200
   ```
4. If the screen terminal is blank, press Enter and then you should see the login prompt.

   The default username and password are both "mendel".

When you're done, exit the `screen` session by pressing Control+A, D.

Update or flash the Dev Board

Periodically, we'll release updates for our Coral software or a new version of Mendel Linux for the Dev Board. This page describes how to install these updates on your board.

Update your board with apt-get

To update the software packages in your current Mendel version—such as when we release a new Edge TPU Runtime and other API libraries—connect to the board (with MDT or the serial console) and run the following commands (first, make sure your board is online).

Note: To upgrade the Mendel version, you must flash a new system image.

```
sudo apt-get update

sudo apt-get dist-upgrade

sudo reboot now
```

Rebooting isn't always required, but recommended in case there are any kernel updates.

Notice: Due to a known issue in Mendel, if your Dev Board is currently running Mendel 2.0 or lower, some packages do not properly upgrade when using `apt-get`. Check your version by running `cat /etc/mendel_version`; if it's 2.0 or lower, we recommend that you flash the system image.

Flash a new system image

Flashing your board is necessary if you want to upgrade to the latest Mendel version or if your board is in a failed state and you want to start clean. If you just want to update your existing system with new packages, instead update with apt-get.

You can see your Mendel version if you connect to the board and run `cat /etc/mendel_version`. Then see what Mendel versions are available on the Software page.

If it's your first time flashing the board (or you're okay with deleting all user data), instead follow the guide to Get started with the Dev Board, which flashes the board using a microSD card.

Note: The flashing procedure below works with Linux and Mac only.

First-time setup

If it's your first time flashing a Dev Board from the command line, start with the following one-time setup:

1. Install the fastboot tool.

   On Linux, you can install as follows:

   ```
   sudo apt-get install fastboot
   ```

   For Mac, it's available from the Android SDK platform tools. This package has many tools, but you only need fastboot. So move that to a location in your `PATH` environment variable, such as `/Users/yourname/bin`.

2. If you're on Linux, add the udev rules required for fastboot:

   ```
   sudo sh -c "echo 'SUBSYSTEM==\"usb\", ATTR{idVendor}==\"0525\", MODE=\"0664\", \
   GROUP=\"plugdev\", TAG+=\"uaccess\"' >> /etc/udev/rules.d/65-edgetpu-board.rules"

   sudo udevadm control --reload-rules && sudo udevadm trigger
   ```

   Also make sure your Linux user account is in the `plugdev` and `dialout` system groups by running this command:

```
sudo usermod -aG plugdev,dialout <username>
```

Then reboot your computer for the new groups to take effect.

Flash the board

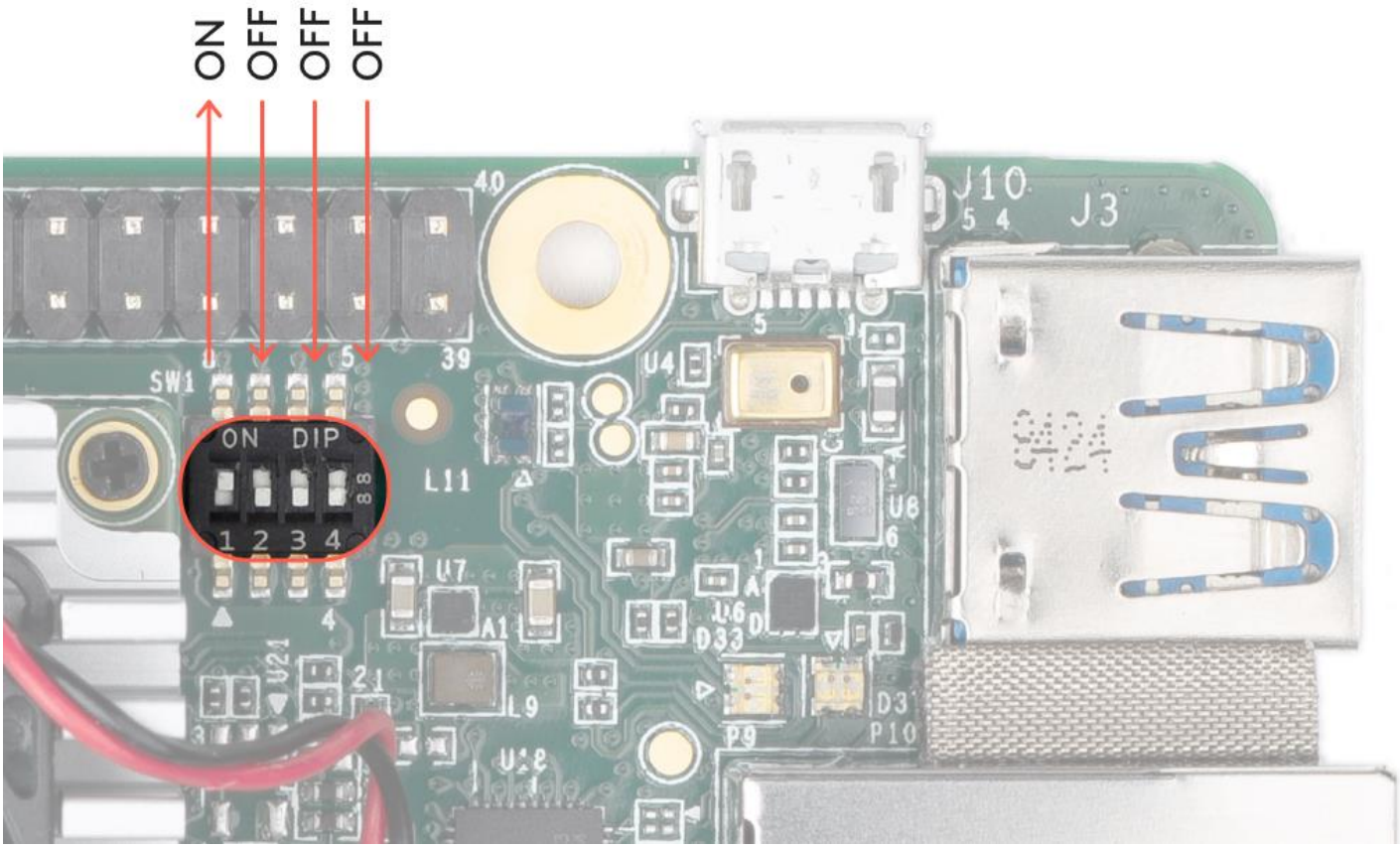
Now you can flash the Dev Board as follows.

Caution: This deletes all system and local data. That is, unless your board is already running Mendel 5.0 or higher, which provides a separate partition for the `/home` directory. So if your board is running Mendel 4.0 or lower, back up any personal data before you proceed. For example, you can create a TAR of your `/home` directory and use `mdt pull` to copy the TAR from the board. Then use `mdt push` to move it back.

1. **Verify the boot mode.**

   First, make sure the boot mode switches are set to eMMC mode as follows. If they're not set this way, power off the board before you change them.

Boot mode	Switch 1	Switch 2	Switch 3	Switch 4
eMMC	ON	OFF	OFF	OFF



1. **Figure 1.** Boot switches set to eMMC mode
2. **Connect to the USB data port.**

Attach a USB cable from your host computer to the USB port on the Dev Board labeled "OTG" (see figure 2).

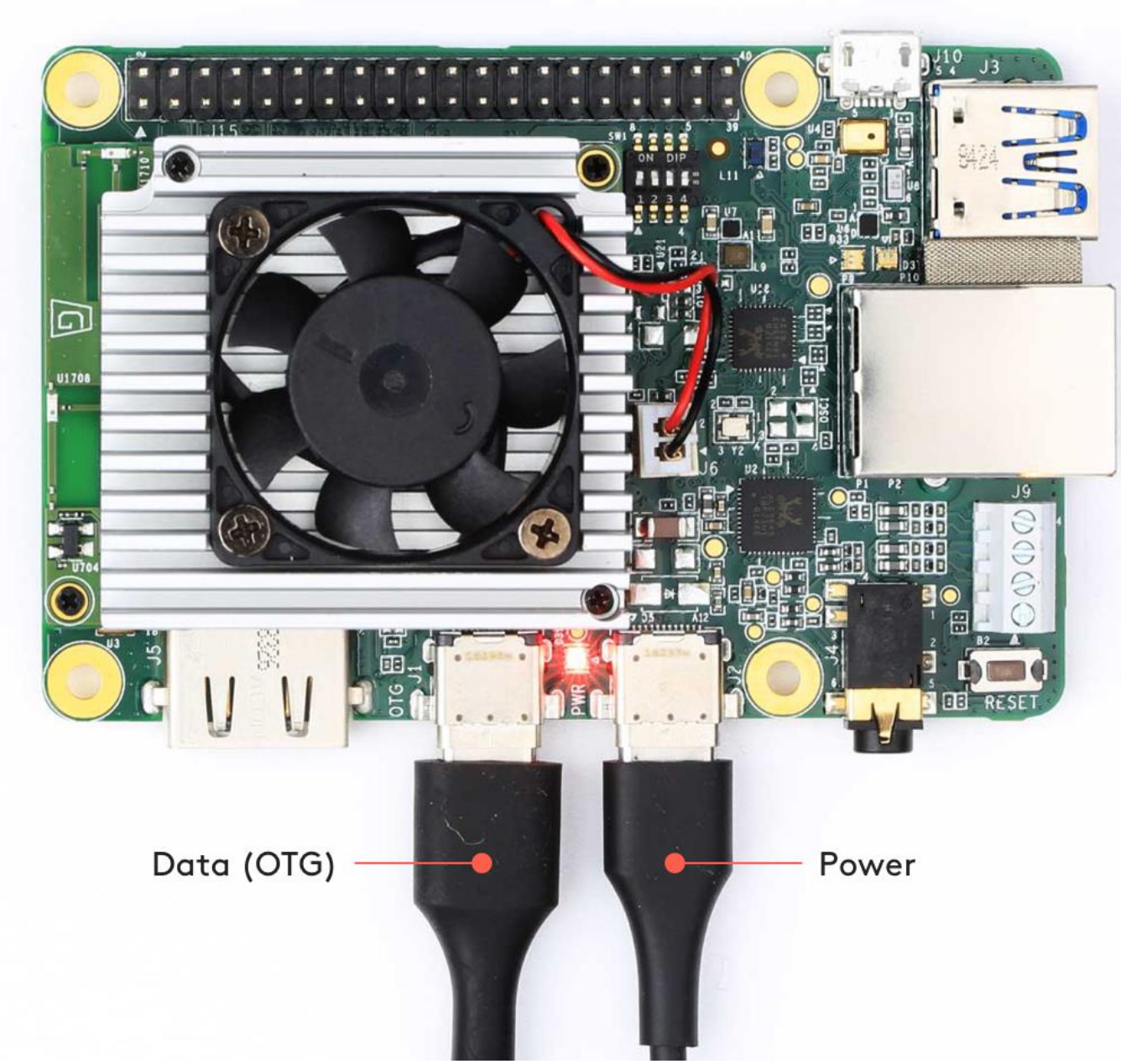


Figure 2. A USB-C data cable connected to the board (in addition to the power cable)

3. **Power on the board.**

   If it's not already booted, plug in the board and wait for it to power on.
   Verify the board is connected to your computer by running this command from your computer (requires MDT):

```
mdt devices
```

   You should see output such as this:

```
orange-horse          (192.168.100.2)
```

4. **Reboot the board into fastboot mode.**
   Caution: You might be about to delete personal data on the board. Now is the time to back it up.

   Run the following command from your computer:

```
mdt reboot-bootloader
```

After a moment, you can verify the board started in fastboot mode with this command (requires `fastboot`):

```
fastboot devices
```

You should see a line printed like this:

```
1b0741d6f0609912        fastboot
```

Help! If `fastboot devices` prints nothing, wait a few more seconds for the device to reboot into fastboot, then try again.
If it still prints nothing, verify the Dev Board is connected to your computer via USB as shown in figure 2 and you rebooted the board with the command `sudo reboot-bootloader`. If so, try installing a more recent version of `fastboot` from Android SDK Platform-tools and try again. (Be sure to add the new fastboot to your `PATH` environment variable.)

5. **Download the system image.**
   Run the following commands **on your host computer**:

```
cd $HOME/Downloads

curl -O https://mendel-linux.org/images/enterprise/eagle/enterprise-eagle-20211117215217.zip

unzip enterprise-eagle-20211117215217.zip \
&& cd enterprise-eagle-20211117215217
```

6. **Flash the board.**
   If your board is currently running Mendel 4.0 or lower, this step wipes all the board data, including the `/home` directory. If it's running Mendel 5.0 or higher, then the `/home` directory is preserved by default (you can intentionally wipe it by adding the `-H` flag to the following command).

```
bash flash.sh
```

   This starts the flashing process and you'll see various output.

   It takes about 5 minutes to complete. When flashing is complete, your board reboots.

7. **Log in.**

   You can log in to the board's shell using MDT:

```
mdt shell
```

   Mac users: If you're on macOS 10.15 (Catalina) or higher, this won't work (even if you preserved the board's `/home` directory and previously installed an SSH key, because flashing the board reset its known networks—you might simply need to get the board back your Wi-Fi using the serial console). See the steps to connect MDT on macOS.
   If MDT is unable to find your device, it's probably because the system is still setting up Mendel. This initial setup takes 2-3 minutes after you flash it (subsequent boot times are much faster). Instead of manually retrying, you can run `mdt wait-for-device && mdt shell` and it will connect when the device is ready.
   Then restore any data you backed up and reconnect to the internet, if necessary.

Note: Your board's hostname is randomly generated the first time it boots from a new flashing. We do this to ensure that each device within a local fleet is likely to have a unique name. Of course, you can change this name using standard Linux hostname tooling (such as `hostname`).
If you attempt to connect with the serial console, the login and password are reset to the defaults: login is **`mendel`**; password is **`mendel`**.

Flash from U-Boot on an SD card

If you can't even boot your board into U-Boot, then you can recover the system by accessing the U-Boot prompt from an image on the SD card, and then enable fastboot mode to flash the board. This also gives you general access to U-Boot for other troubleshooting.

Tip: To just delete everything and re-install Mendel, follow the new flash procedure in the Dev Board get started guide, which automatically flashes the board using an SD card (no command line needed).

1. **Enable boot from SD card.**

   Unplug the Dev Board and change the boot mode switches to boot from SD card:

Boot mode	Switch 1	Switch 2	Switch 3	Switch 4
SD Card	ON	OFF	ON	ON

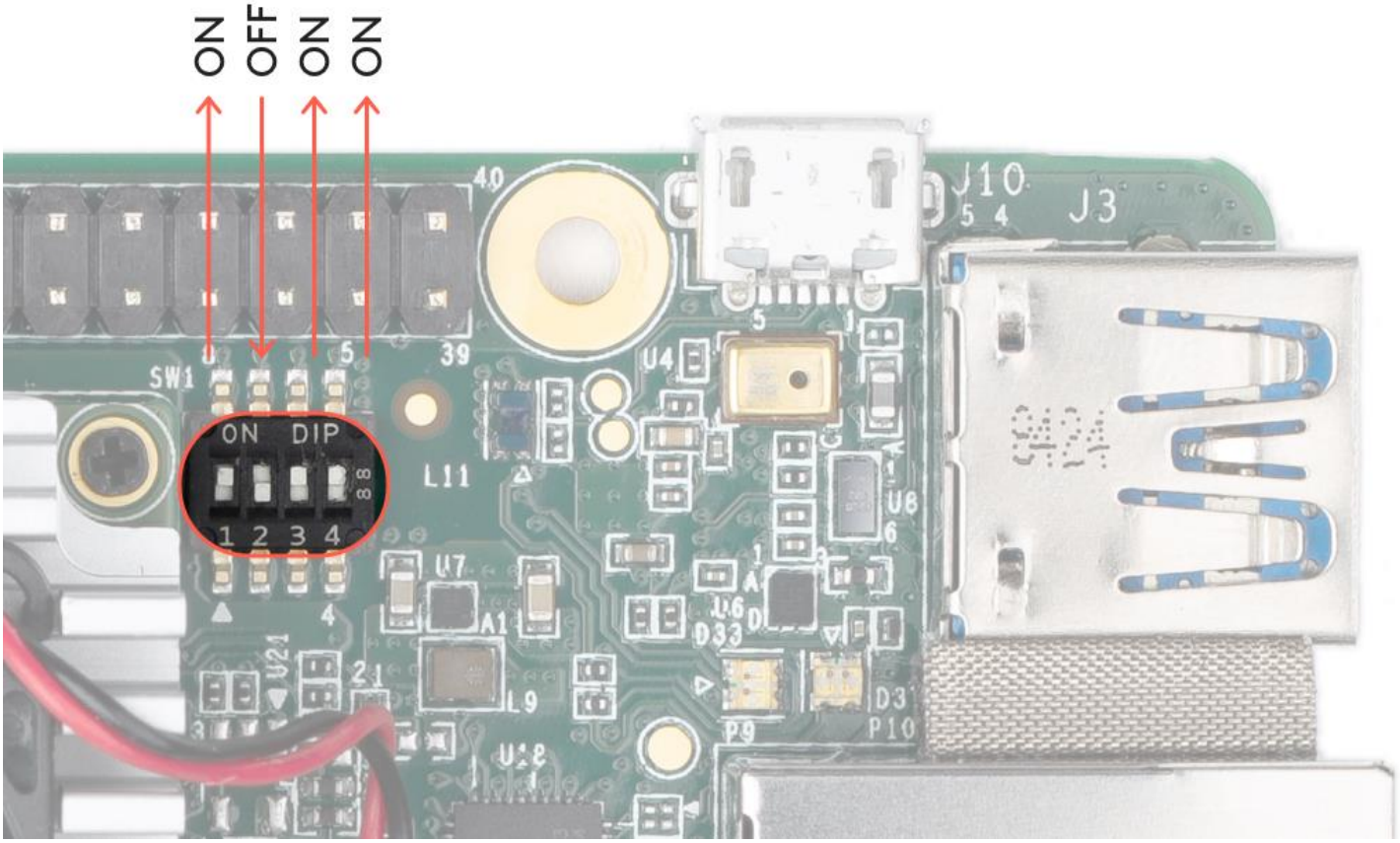


Figure 3. Boot switches set to SD card mode

2. **Flash an SD card with the recovery image.**

   On your host computer, download and unpack the latest system image:

```
curl -O https://mendel-linux.org/images/enterprise/eagle/enterprise-eagle-20211117215217.zip

unzip enterprise-eagle-20211117215217.zip
```

   Inside the package, find the `recovery.img` file.

   Then use a program such as balenaEtcher to flash the `recovery.img` file to your microSD card.

   Help! If the flash program displays a warning such as, "The image does not appear to contain a partition table," that's okay—it's true that there is no partition table in this image, but this is working as intended so you can continue.

3. **Connect to the serial console.**

   Note: The board should not be powered on yet.

Use a USB-micro-B cable to connect your host computer to the serial console port on the board (see figure 4). The orange and green LEDs on the board will illuminate.
Follow these instructions to open the serial console (complete all steps, but **do not** plug the board into power). When you complete those steps, the prompt should disappear and your terminal should become completely blank (because the board is not powered on yet).

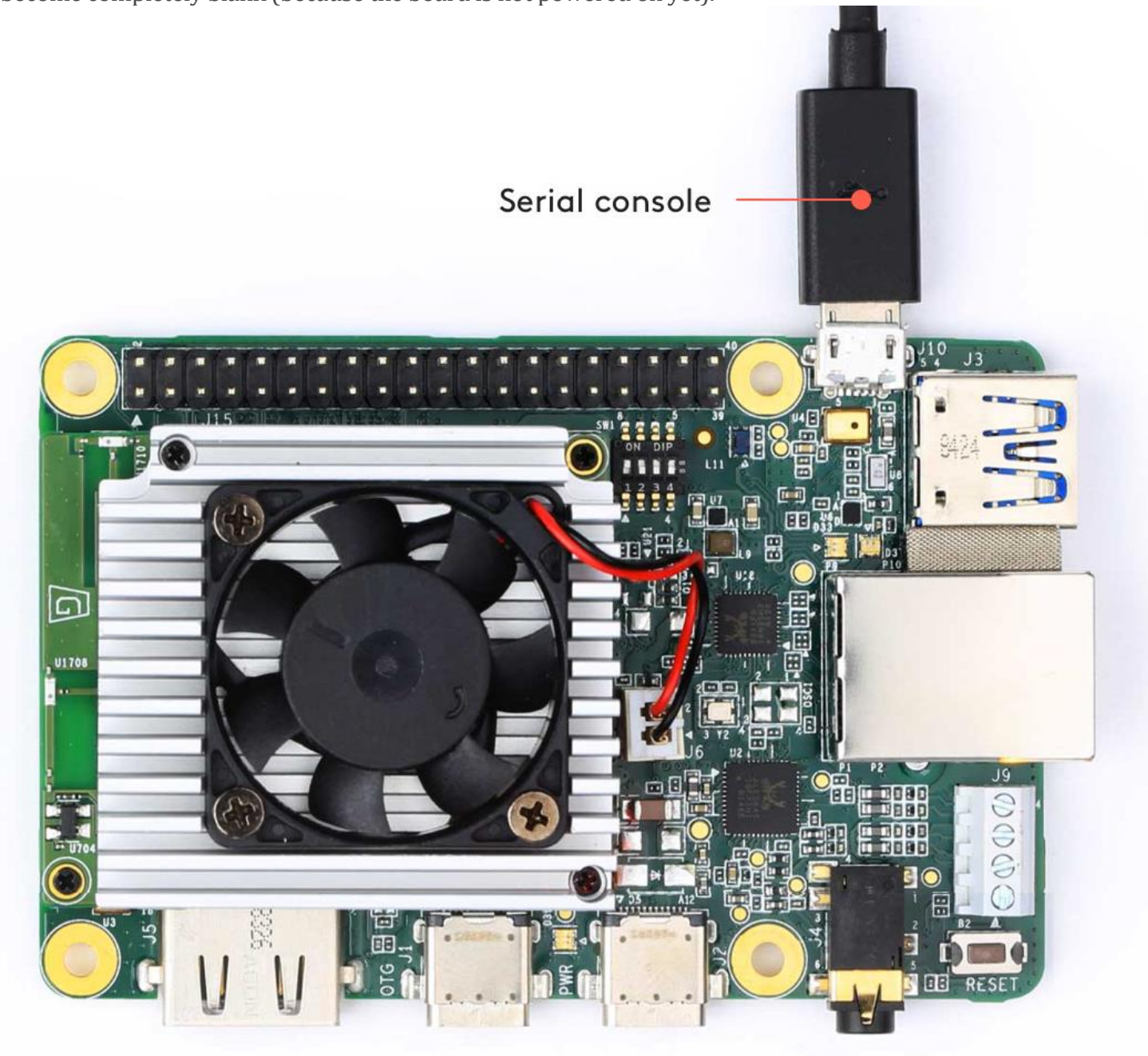


Figure 4. A USB-micro-B cable connected to the serial console port

4. **Insert the SD card and then power on the board.**

   The SD card slot is on the bottom of the Dev Board.

   Plug in your 2 - 3A power cable to the USB-C port labeled "PWR" (see figure 5).
   Caution: Do not attempt to power the board by connecting it to your computer.

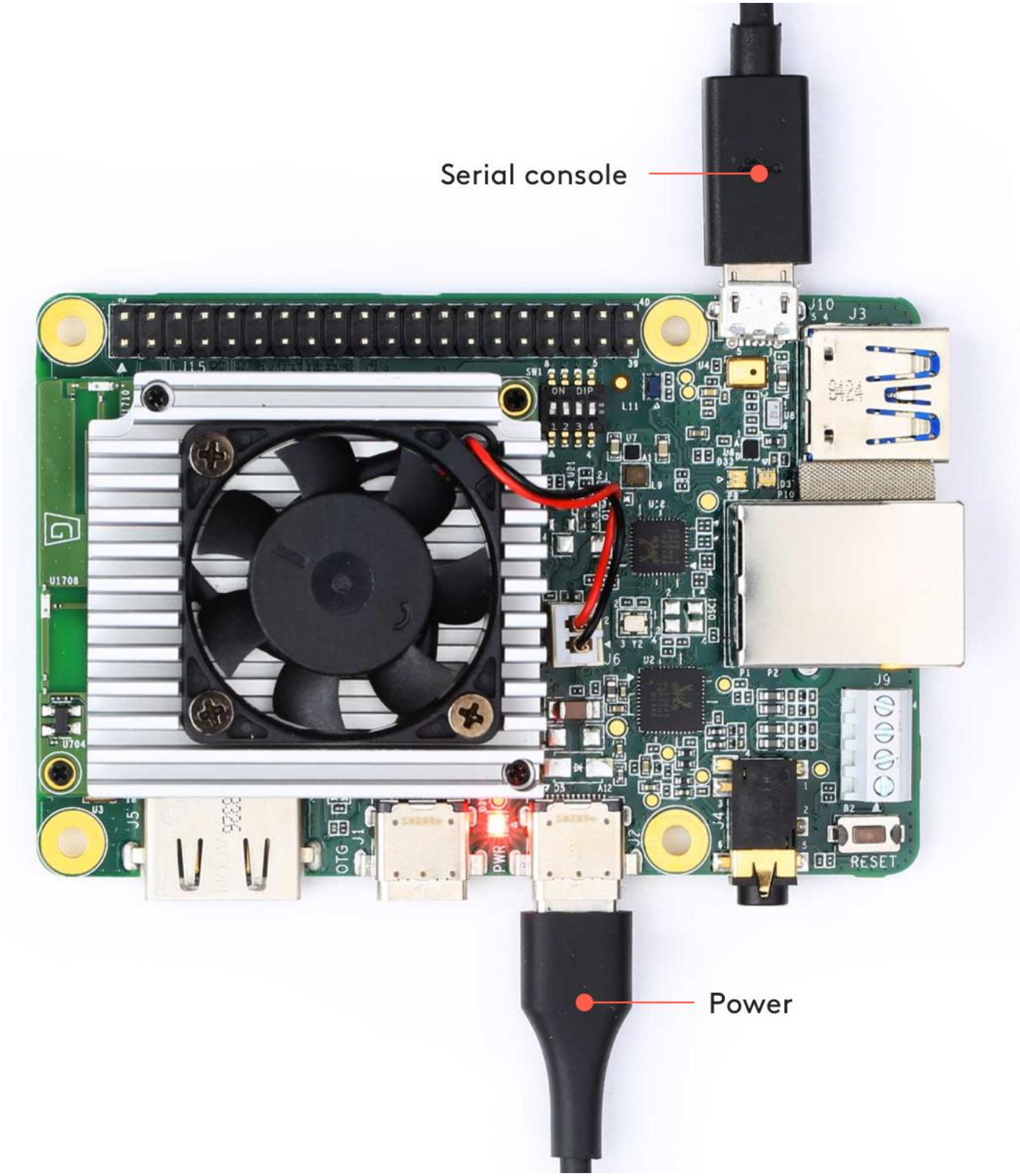


Figure 5. A USB-C power cable connected to the board (in addition to the serial cable)

5. **Start fastboot.**
   At the `u-boot` prompt that appears in your `screen` session (from step 3), start fastboot mode:

   ```
   fastboot 0
   ```

   The cursor moves to the next line but there is no output. That's as expected.
6. **Connect to the USB data port.**

Use a USB-C cable to connect your host computer (Linux or Mac) to the USB-C data port on the Dev Board labeled "OTG" (see figure 6). (This is the connection used by fastboot.)

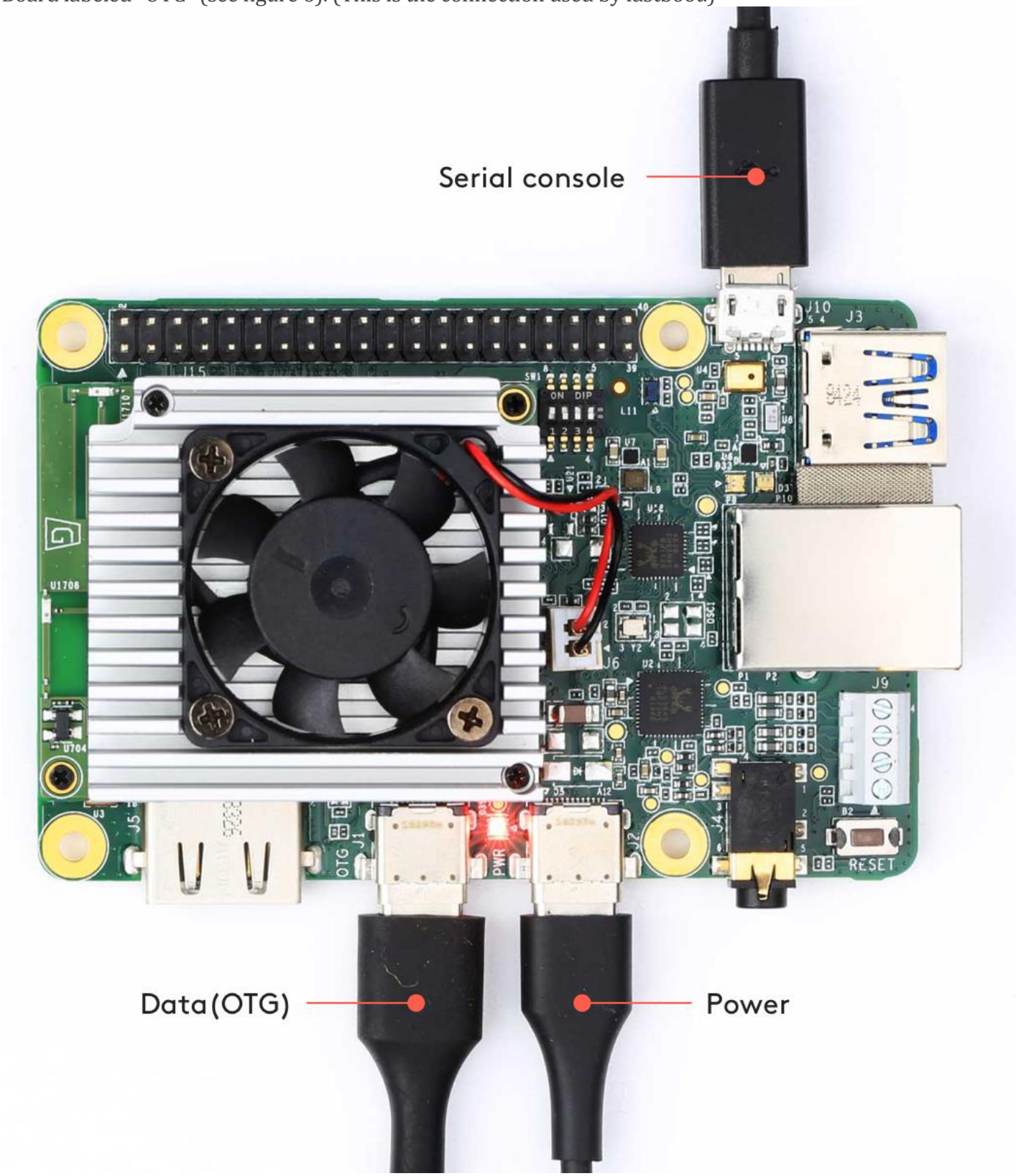


Figure 6. A USB-C data cable connected to the board (in addition to the serial and power cables)

After a moment, you can verify that the board started in fastboot mode by running the following command **from your connected host computer** (requires `fastboot`):

```
fastboot devices
```

You should see a line printed like this:

```
1b0741d6f0609912        fastboot
```

7. **Flash the image.**
   On your host computer, navigate into the unpacked system image directory (from step 2) and execute the flash script:

```
cd enterprise-eagle-20211117215217

bash flash.sh
```

When flashing is complete, your board will reboot. However, because you set the boot mode to SD card, that's what it will do so you'll again see the `u-boot` prompt. So unplug the power and reset the boot switches to eMMC mode:

Boot mode	Switch 1	Switch 2	Switch 3	Switch 4
eMMC	ON	OFF	OFF	OFF

Then boot the board again. When it's done booting, the console prompts you to login.
Login is **`mendel`**
Password is **`mendel`**
Tip: You can instead open a shell using MDT.
Once logged in, be sure to download other software updates with the following commands (be sure your board is online first):

```
sudo apt-get update

sudo apt-get dist-upgrade
```

Flash a new board

If you just unboxed your Dev Board, we recommend that you follow the get started guide, which shows how to flash the board using a microSD card. However, if you don't have a suitable microSD card, then you can instead manually flash a new board as follows.
Caution: This flashing procedure is significantly more complicated than using a microSD card, so you might be happier if you instead take the time to find a microSD card and follow the get started guide.

1. Gather the requirements

Before you begin, collect the following hardware:

- A host computer running Linux or Mac
  - Python 3 installed
- One USB-C power supply (2 A / 5 V), such as a phone charger
- One USB-C to USB-A cable
- One USB-micro-B to USB-A cable—be sure this cable supports data transfer (not just power)

You also need the following software tools on your host computer:

- A serial console program such as `screen`, `picocom`, or PuTTY (among many others).
  Our instructions use `screen` and it's available on Mac computers by default.
  If you're on Linux and don't already prefer another program, we suggest you install `screen` as follows:

```
sudo apt-get install screen
```

- The latest `fastboot` tool.

  On Linux, you can install as follows:
  ```
  sudo apt-get install fastboot
  ```
  For Mac, it's available from the Android SDK platform tools. This package has many tools, but you only need fastboot. So move that to a location in your `PATH` environment variable, such as `/Users/yourname/bin`.

  Now verify it works:
  ```
  fastboot --version
  ```
  For Mac compatibility, the version must be 28.0.2 or higher.

Also configure your system to communicate with the board:

- **On Linux:**

  Run the following commands to add the udev rule required for fastboot:
  ```
  sudo sh -c "echo 'SUBSYSTEM==\"usb\", ATTR{idVendor}==\"0525\", MODE=\"0664\", \
  GROUP=\"plugdev\", TAG+=\"uaccess\"' >> /etc/udev/rules.d/65-edgetpu-board.rules"

  sudo udevadm control --reload-rules && sudo udevadm trigger
  ```
  Also make sure your Linux user account is in the `plugdev` and `dialout` system groups by running this command:
  ```
  sudo usermod -aG plugdev,dialout <username>
  ```
  Then reboot your computer for the new groups to take effect.
- **On Mac:**
  Caution: Before installing the following package, be sure you've applied all available macOS software updates. Otherwise, you might be blocked from installing due to system security that disables the **Allow** button in System Preferences.
  Install the CP210x USB to UART Bridge Virtual COM Port (VCP) driver for Mac.

2. Enable the correct boot mode

Before you begin the flashing procedure, verify the following:

1. The board is completely unplugged (not powered and not connected to your computer).
2. The boot mode switches are set to eMMC mode (see figure 7):

Boot mode	Switch 1	Switch 2	Switch 3	Switch 4
eMMC	ON	OFF	OFF	OFF

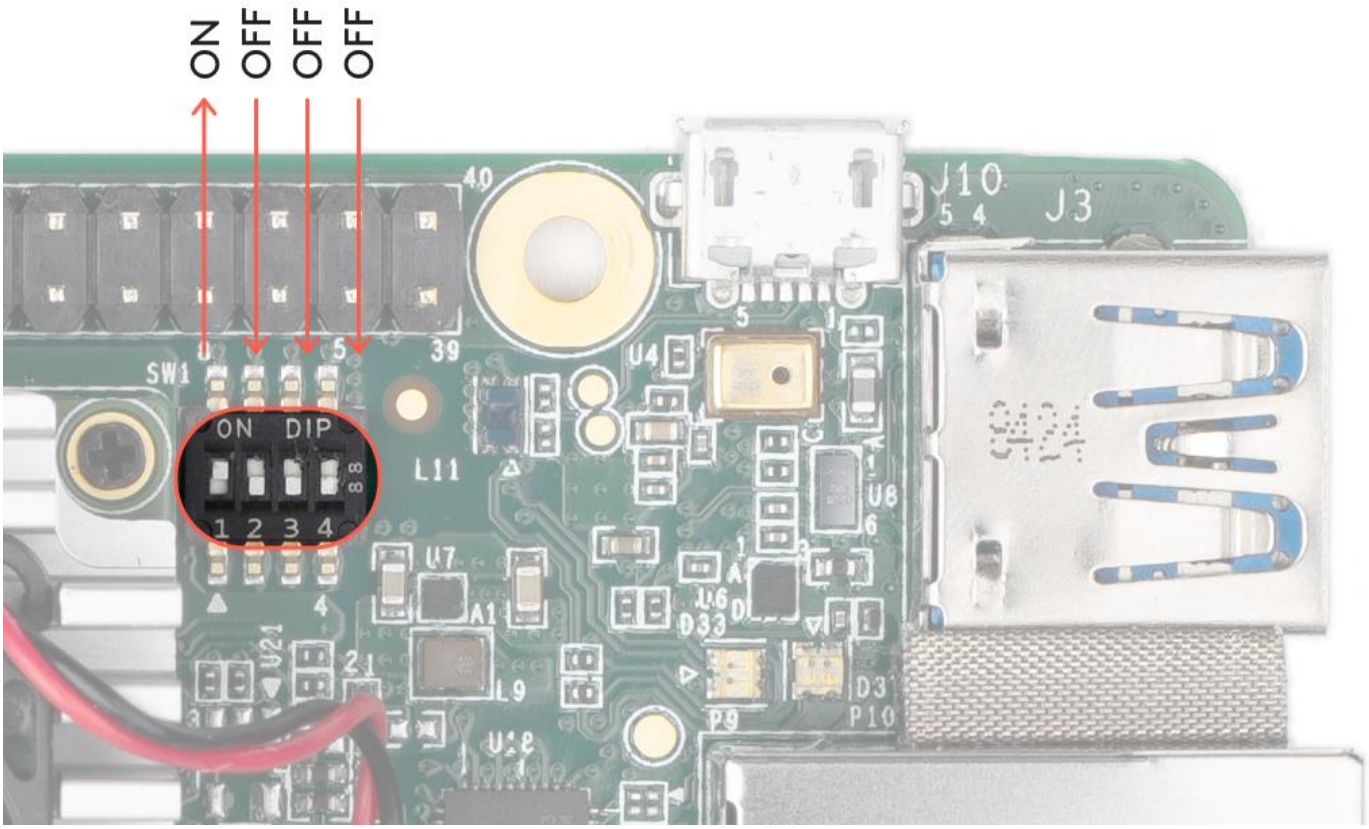


Figure 7. Boot switches set to eMMC mode
Note: Do not power the board or connect any cables until instructed to do so.

3: Initiate fastboot mode

If your Dev Board was manufactured before April 10, 2019, then you need to initiate fastboot on the board.

You can determine the manufactured date from the serial number etched onto the heat sink, as shown in figure 8.

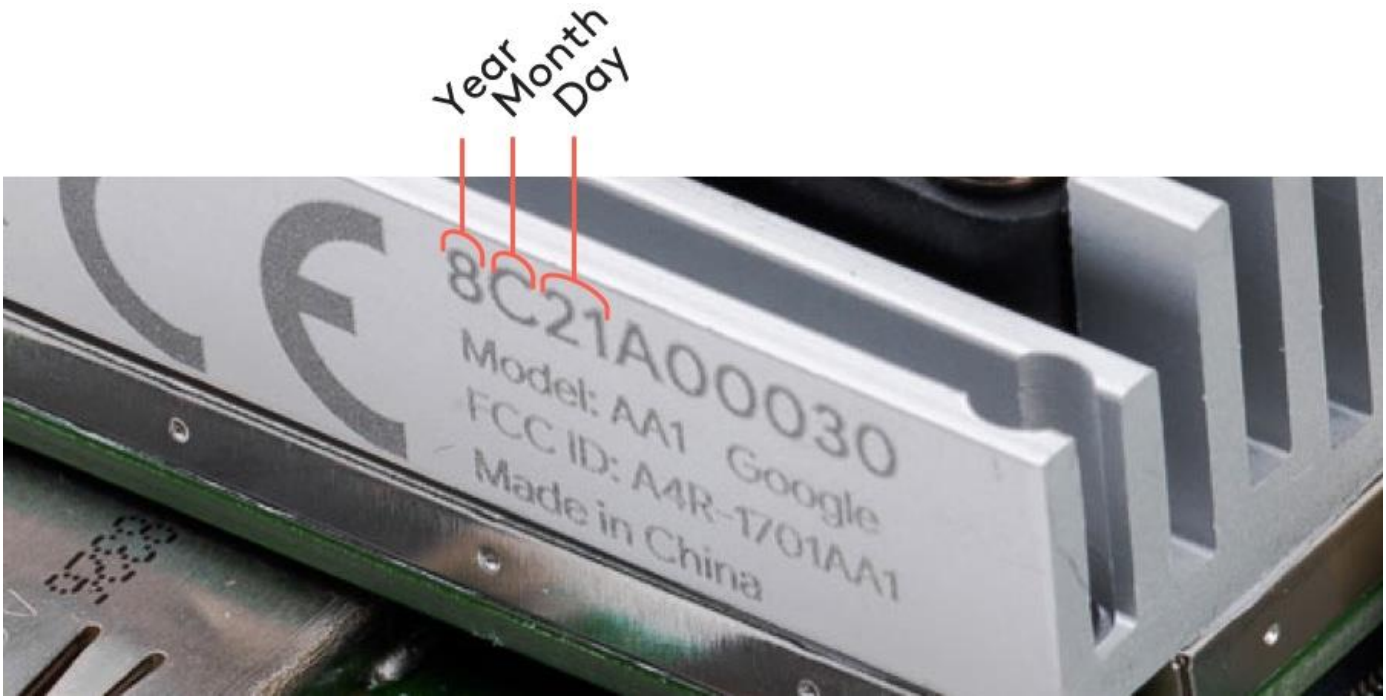


Figure 8. The manufactured date in the serial number

Field	Description
Year	Last digit of the year of manufacture: 0 - 9
Month	Month of manufacture: 1 to 9 (Jan to Sep), A (Oct), B (Nov), C (Dec)
Day	Day of manufacture: 01 to 31 (1st to 31st)

If your serial number starts with "9410" or higher, then it was manufactured on April 10th, 2019, or later, so you can skip to execute the flash script (because your board automatically boots into fastboot mode).

But if the date reads "9409" or lower, then you need to perform the following steps to enable fastboot mode:

1. **Connect the serial console cable.**

   Use your micro-B USB cable to connect your host computer to the serial console port on the board (see figure 9). The orange and green LEDs on the board should illuminate.
   Note: The board should not be powered on yet.

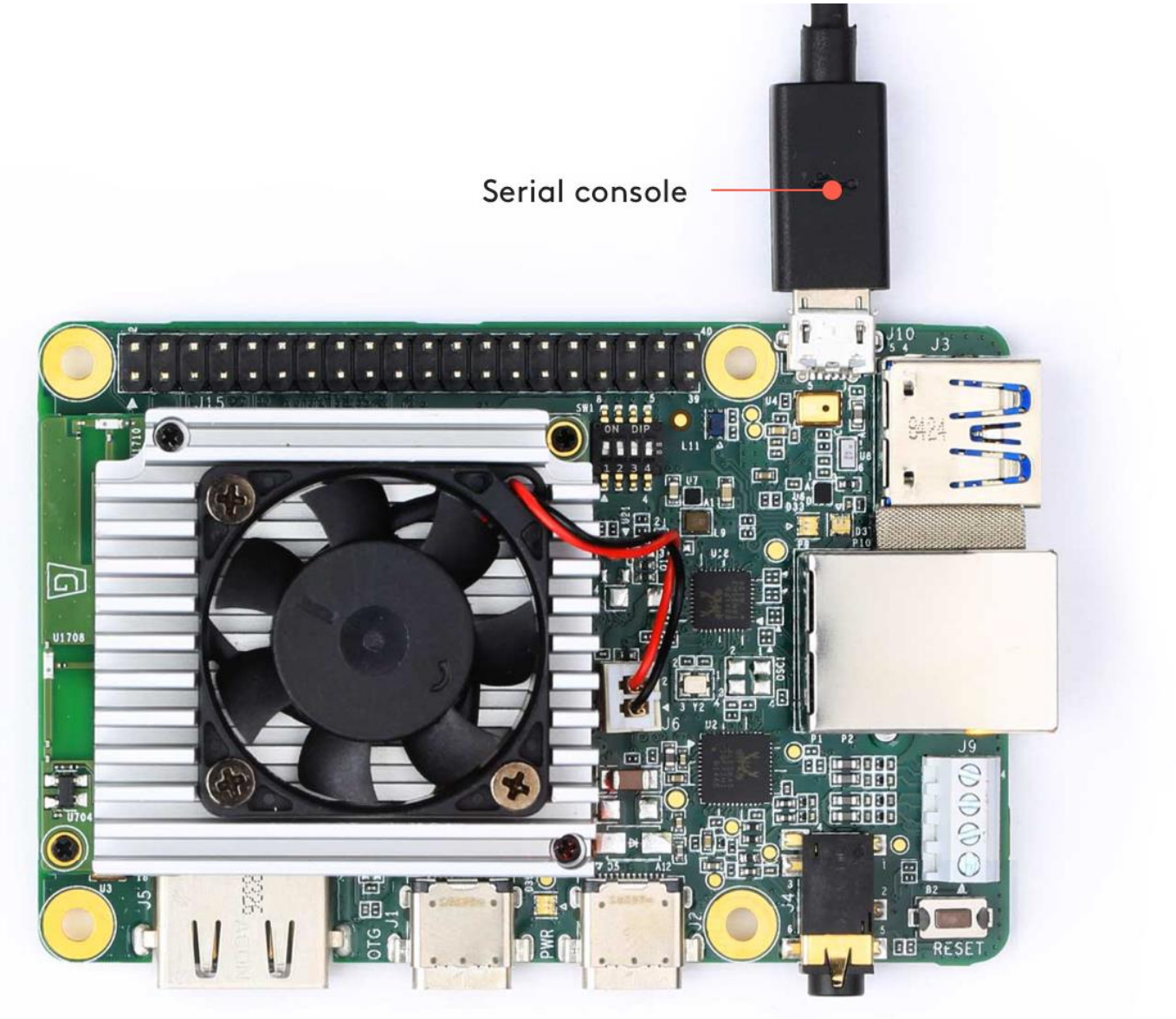


Figure 9. A micro-B USB cable connected to the serial console port

2. **Initiate the serial console.**

   Open a terminal on your host computer and start the serial console as follows:

   - **On Linux:**

     Determine the device filename for the serial connection by running this command on your Linux computer:

     ```
     dmesg | grep ttyUSB
     ```

     You should see two results such as this:

     ```
     [ 6437.706335] usb 2-13.1: cp210x converter now attached to ttyUSB0
     [ 6437.708049] usb 2-13.1: cp210x converter now attached to ttyUSB1
     ```

     If you don't see results like this, double-check your USB cable.
     Then use the name of the *first* filename listed as a `cp210x converter` to open the serial console connection (this example uses `ttyUSB0` as shown from above):

     ```
     screen /dev/ttyUSB0 115200
     ```

     Help!

     If you see `[screen is terminating]`, it might also be due to the above missing groups, or there's something else wrong with `screen`—ensure all `screen` sessions are closed (type `screen -ls` to see open sessions), unplug the USB cable from the Dev Board, and then try again.

- **On Mac:**

  First, verify the board is visible by running this command:

  ```
  ls /dev/cu*
  ```

  You should see `/dev/cu.SLAB_USBtoUART` listed. If not, either there's a problem with your USB cable or the driver is not loaded. You can load the driver with `sudo kextload /Library/Extensions/SiLabsUSBDriver.kext` and then go to the system *Security & Privacy* preferences and click *Allow*. You also might need reboot your computer.

  Then connect with this command:

  ```
  screen /dev/cu.SLAB_USBtoUART 115200
  ```

  If this does not connect, check the `ls /dev/cu*` output again—you might need to instead use the device name with a number at the end.

The prompt should disappear and your terminal should become completely blank. That's expected, because you've established a connection but the board is not turned on yet.

3. **Power the board.**

   Plug in your 2 - 3 A power cable to the USB-C port labeled "PWR" (see figure 10).
   Caution: Do not attempt to power the board by connecting it to your computer.

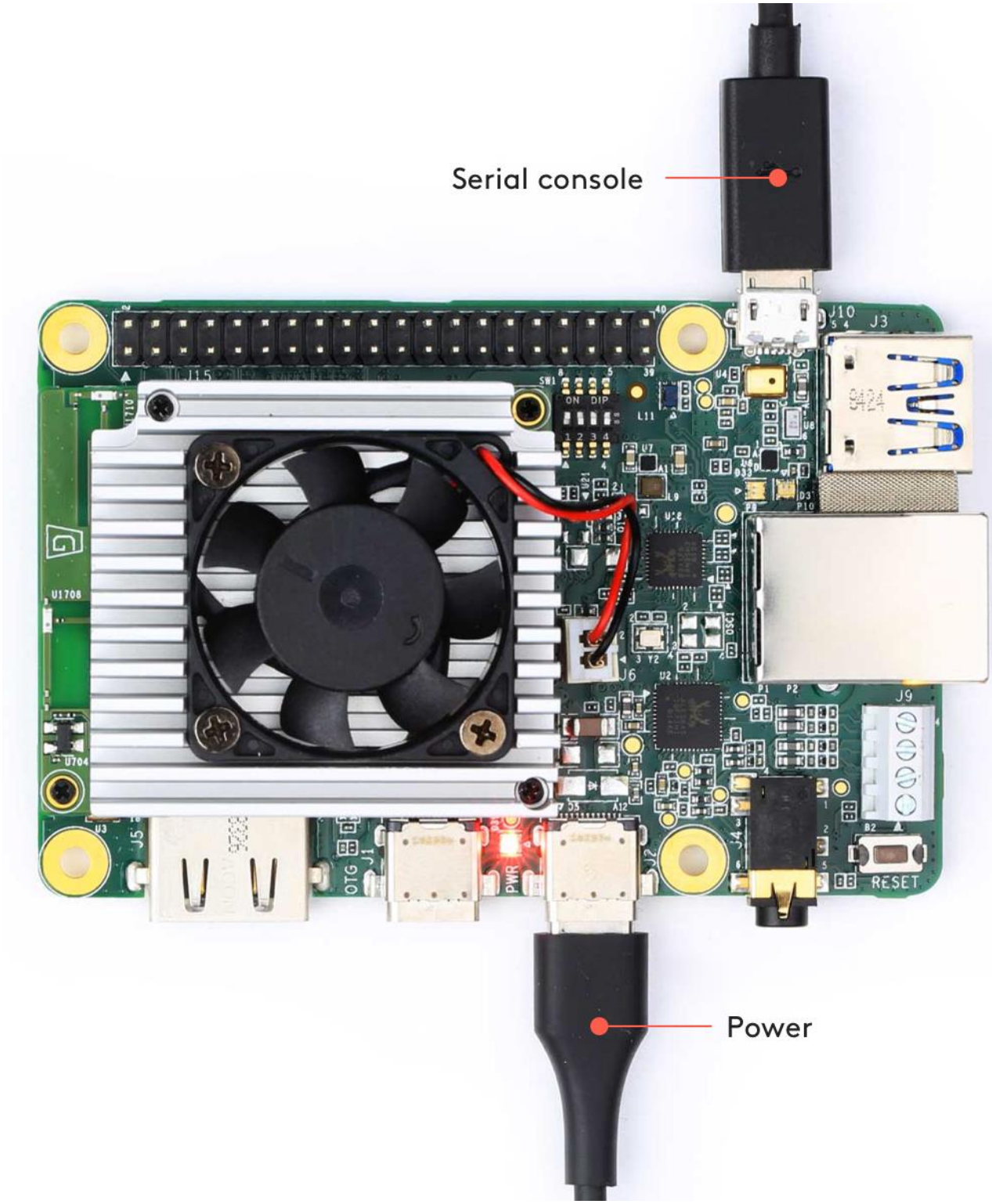


Figure 10. A USB-C power cable connected to the board (in addition to the serial cable)

The board's red LED will illuminate and the fan might turn on.
Help! If you still don't see anything in the serial console screen, press Enter.
Your serial console (the `screen` terminal) should arrive at the `u-boot` prompt. You should see a "Welcome" message that tells you to visit g.co/coral/setup, which brings you to this page. So you're all good; you can continue.
Note: If you instead see a long stream of messages, followed by a login prompt, then your board is already flashed with a system image. You can either disconnect the serial console and connect to the shell with MDT or if you still want to reflash the board, then instead read flash a new system image.

4. **Start fastboot.**
   In your serial console's `u-boot` prompt, execute the following:

   ```
   fastboot 0
   ```

   The cursor should simply move to the next line. Fastboot is now waiting for the host to begin flashing a system image.
5. **Disconnect the serial console.**

   You won't need the serial console connection anymore, so you can close the terminal and remove the micro-B USB cable.

4: Execute the flash script

Now you're ready to flash the board.

1. **Connect the USB-C cable.**
   Use your USB-C cable to connect your host computer to the USB-C data port labeled "OTG" on the Dev Board. (If you skipped the above procedure to initiate fastboot, then you also need to plug in your 2 - 3A power cable to the USB-C port labeled "PWR" and wait a few moments for the board to power on.)

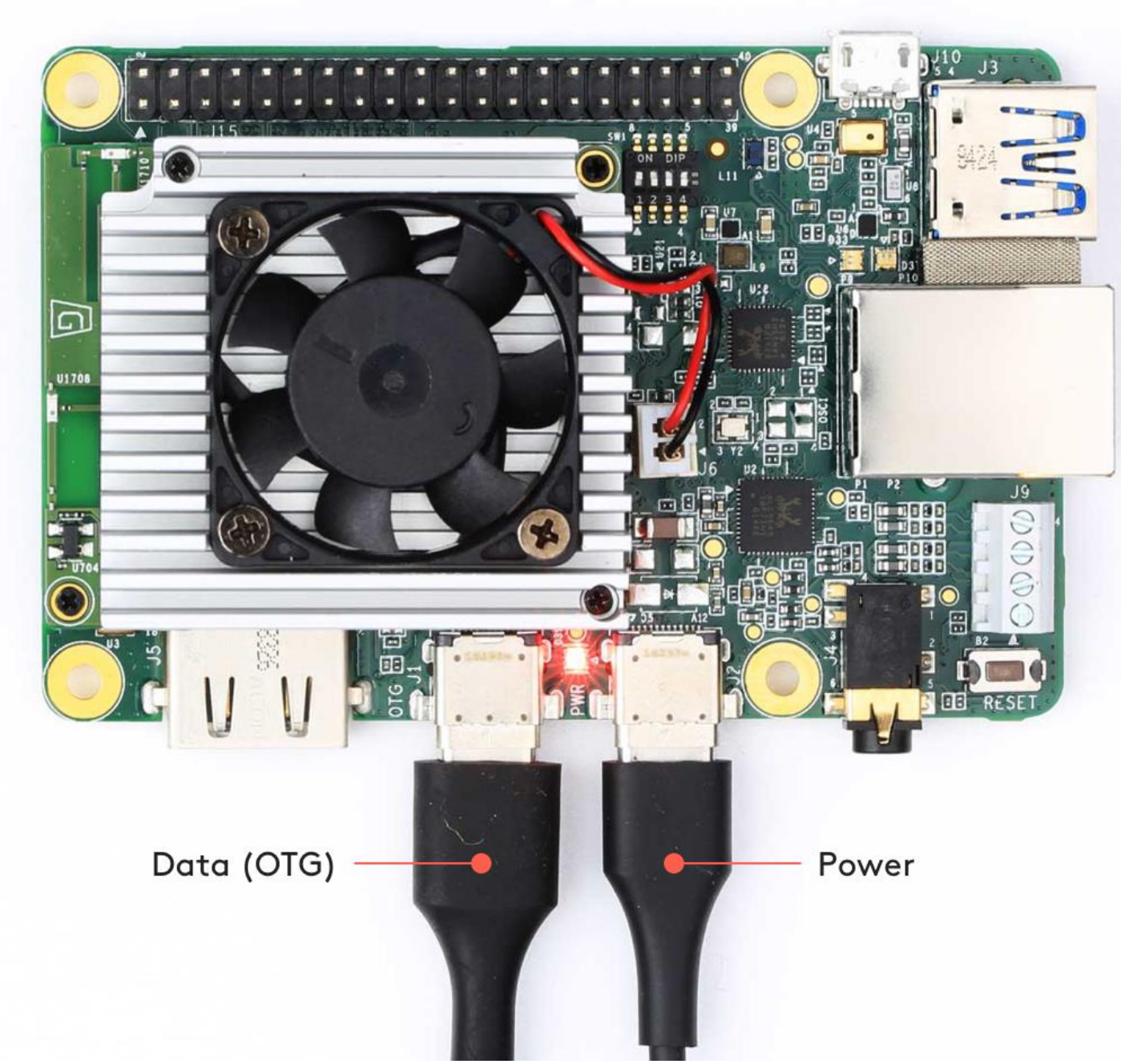


Figure 11. A USB-C data cable and power cable connected to the board

2. **Verify fastboot sees your device.**

   Open a new terminal on your host computer and execute the following command:

   ```
   fastboot devices
   ```

   You should see a line printed like this (your numbers will be different):

   ```
   1b0741d6f0609912        fastboot
   ```

   If you don't see anything printed, be sure you have the latest version of `fastboot` (as per the requirements above). If you recently updated fastboot but it's still blank, then repeat the steps to initiate fastboot mode.

3. **Download and flash the system image.**

   From the same terminal, execute the following:

   ```
   cd ~/Downloads
   ```

```
curl -O https://mendel-linux.org/images/enterprise/eagle/enterprise-eagle-20211117215217.zip

unzip enterprise-eagle-20211117215217.zip \
&& cd enterprise-eagle-20211117215217

bash flash.sh
```

This starts the flashing process and you'll see various output.

It takes about 5 minutes for flashing to complete. When it's done, the board reboots and your terminal prompt returns to you.

TensorFlow models on the Edge TPU

In order for the Edge TPU to provide high-speed neural network performance with a low-power cost, the Edge TPU supports a specific set of neural network operations and architectures. This page describes what types of models are compatible with the Edge TPU and how you can create them, either by compiling your own TensorFlow model or retraining an existing model with transfer-learning.
If you're looking for information about how to run a model, read the Edge TPU inferencing overview.
Or if you just want to try some models, check out our trained models.

Compatibility overview

The Edge TPU is capable of executing deep feed-forward neural networks such as convolutional neural networks (CNN). It supports only TensorFlow Lite models that are fully 8-bit quantized and then compiled specifically for the Edge TPU.
If you're not familiar with TensorFlow Lite, it's a lightweight version of TensorFlow designed for mobile and embedded devices. It achieves low-latency inference in a small binary size—both the TensorFlow Lite models and interpreter kernels are much smaller. TensorFlow Lite models can be made even smaller and more efficient through quantization, which converts 32-bit parameter data into 8-bit representations (which is required by the Edge TPU).
You cannot train a model directly with TensorFlow Lite; instead you must convert your model from a TensorFlow file (such as a `.pb` file) to a TensorFlow Lite file (a `.tflite` file), using the TensorFlow Lite converter.
Figure 1 illustrates the basic process to create a model that's compatible with the Edge TPU. Most of the workflow uses standard TensorFlow tools. Once you have a TensorFlow Lite model, you then use our Edge TPU compiler to create a `.tflite` file that's compatible with the Edge TPU.

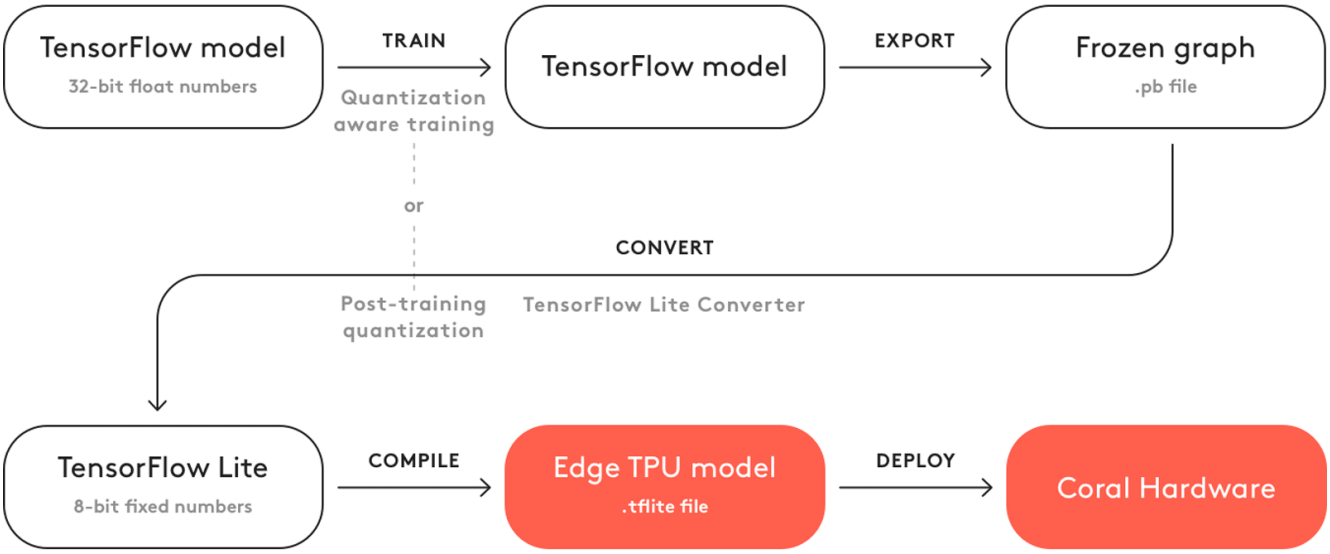


Figure 1. The basic workflow to create a model for the Edge TPU

However, you don't need to follow this whole process to create a good model for the Edge TPU. Instead, you can leverage existing TensorFlow models that are compatible with the Edge TPU by retraining them with your own dataset. For example, MobileNet is a popular image classification/detection model architecture that's compatible with the Edge TPU. We've created several versions of this model that you can use as a starting point to create your own model that recognizes different objects. To get started, see the next section about how to retrain an existing model with transfer learning.
If you have designed—or plan to design—your own model architecture, then you should read the section below about model requirements.

Transfer learning

Instead of building your own model and then training it from scratch, you can retrain an existing model that's already compatible with the Edge TPU, using a technique called transfer learning (sometimes also called "fine tuning").

Training a neural network from scratch (when it has no computed weights or bias) can take days-worth of computing time and requires a vast amount of training data. But transfer learning allows you to start with a model that's already trained for a related task and then perform further training to teach the model new classifications using a smaller training dataset. You can do this by retraining the whole model (adjusting the weights across the whole network), but you can also achieve very accurate results by simply removing the final layer that performs classification, and training a new layer on top that recognize your new classes.
Using this process, with sufficient training data and some adjustments to the hyperparameters, you can create a highly accurate TensorFlow model in a single sitting. Once you're happy with the model's performance, simply convert it to TensorFlow Lite and then compile it for the Edge TPU. And because the model architecture doesn't change during transfer learning, you know it will fully compile for the Edge TPU (assuming you start with a compatible model).
To get started without any setup, try our Google Colab retraining tutorials. All these tutorials perform transfer learning in cloud-hosted Jupyter notebooks.

Transfer learning on-device

If you're using an image classification model, you can also perform accelerated transfer learning on the Edge TPU. Our Python and C++ APIs offer two different techniques for on-device transfer learning:

- Weight imprinting on the last layer (`ImprintingEngine` in Python or C++)
- Backpropagation on the last layer (`SoftmaxRegression` in Python or C++)

In both cases, you must provide a model that's specially designed to allow training on the last layer. The required model structure is different for each API, but the result is basically the same: the last fully-connected layer where classification occurs is separated from the base of the graph. Then only the base of the graph is compiled for the Edge TPU, which leaves the weights in the last layer accessible for training. More detail about the model architecture is available in the corresponding documents below. For now, let's compare how retraining works for each technique:

- Weight imprinting takes the output (the embedding vectors) from the base model, adjusts the activation vectors with L2-normalization, and uses those values to compute new weights in the final layer—it averages the new vectors with those already in the last layer's weights. This allows for effective training of new classes with very few sample images.
- Backpropagation is an abbreviated version of traditional backpropagation. Instead of backpropagating new weights to all layers in the graph, it updates only the fully-connected layer at the end of the graph with new weights. This is the more traditional training strategy

that generally achieves higher accuracy, but it requires more images and multiple training iterations.

When choosing between these training techniques, you might consider the following factors:

- **Training sample size:** Weight imprinting is more effective if you have a relatively small set of training samples: anywhere from 1 to 200 sample images for each class (as few as 5 can be effective and the API sets a maximum of 200). If you have more samples available for training, you'll likely achieve higher accuracy by using them all with backpropagation.
- **Training sample variance:** Backpropagation is more effective if your dataset includes large intra-class variance. That is, if the images within a given class show the subject in significantly different ways, such as in angle or size, then backpropagation probably works better. But if your application operates in an environment where such variance is low, and your training samples thus also have little intra-class variance, then weight imprinting can work very well.
- **Adding new classes:** Only weight imprinting allows you to add new classes to the model after you've begun training. If you're using backpropagation, adding a new class after you've begun training requires that you restart training for all classes. Additionally, weight imprinting allows you to retain the classes from the pre-trained model (those trained before converting the model for the Edge TPU); whereas backpropagation requires all classes to be learned on-device.
- **Model compatibility:** Backpropagation is compatible with more model architectures "out of the box"; you can convert existing, pre-trained MobileNet and Inception models into embedding extractors that are compatible with on-device backpropagation. To use weight imprinting, you must use a model with some very specific layers and then train it in a particular manner before using it for on-device training (currently, we offer a version of MobileNet v1 with the proper modifications).

In both cases, the vast majority of the training process is accelerated by the Edge TPU. And when performing inferences with the retrained model, the Edge TPU accelerates everything except the final classification layer, which runs on the CPU. But because this last layer accounts for only a small portion of the model, running this last layer on the CPU should not significantly affect your inference speed.

To learn more about each technique and try some sample code, see the following pages:

- Retrain a classification model on-device with weight imprinting
- Retrain a classification model on-device with backpropagation

Model requirements

If you want to build a TensorFlow model that takes full advantage of the Edge TPU for accelerated inferencing, the model must meet these basic requirements:

- **Tensor parameters are quantized** (8-bit fixed-point numbers; int8 or uint8).
- **Tensor sizes are constant at compile-time** (no dynamic sizes).
- **Model parameters (such as bias tensors) are constant at compile-time**.
- **Tensors are either 1-, 2-, or 3-dimensional**. If a tensor has more than 3 dimensions, then only the 3 innermost dimensions may have a size greater than 1.
- **The model uses only the operations supported by the Edge TPU** (see table 1 below).

Failure to meet these requirements could mean your model cannot compile for the Edge TPU at all, or only a portion of it will be accelerated, as described in the section below about compiling.

Note: If you pass a model to the Edge TPU Compiler that uses float inputs, the compiler leaves a quantize op at the beginning of your graph (which runs on the CPU). So as long as your tensor parameters are quantized, it's okay if the input and output tensors are float because they'll be converted on the CPU.

Supported operations

When building your own model architecture, be aware that only the operations in the following table are supported by the Edge TPU. If your architecture uses operations not listed here, then only a portion of the model will execute on the Edge TPU.
Note: When creating a new TensorFlow model, also refer to the list of operations compatible with TensorFlow Lite.

Table 1. All operations supported by the Edge TPU and any known limitations

Operation name	Runtime version*	Known limitations
Add	All	
AveragePool2d	All	No fused activation function.
Concatenation	All	No fused activation function. If any input is a compile-time constant tensor, there must be only 2 inputs, and this constant tensor must be all zeros (effectively, a zero-padding op).
Conv2d	All	Must use the same dilation in x and y dimensions.
DepthwiseConv2d	≤12	Dilated conv kernels are not supported.
	≥13	Must use the same dilation in x and y dimensions.
ExpandDims	≥13	
FullyConnected	All	Only the default format is supported for fully-connected weights. Output tensor is one-dimensional.
L2Normalization	All	
Logistic	All	
LSTM	≥14	Unidirectional LSTM only.
Maximum	All	
MaxPool2d	All	No fused activation function.
Mean	≤12	No reduction in batch dimension. Supports reduction along x- and/or y-dimensions only.
	≥13	No reduction in batch dimension. If a z-reduction, the z-dimension must be multiple of 4.
Minimum	All	
Mul	All	

Table 1. All operations supported by the Edge TPU and any known limitations

Operation name	Runtime version*	Known limitations
Pack	≥13	No packing in batch dimension.
Pad	≤12	No padding in batch dimension. Supports padding along x- and/or y-dimensions only.
	≥13	No padding in batch dimension.
PReLU	≥13	Alpha must be 1-dimensional (only the innermost dimension can be >1 size). If using Keras PReLU with 4D input (batch, height, width, channels), then shared_axes must be [1,2] so each filter has only one set of parameters.
Quantize	≥13	
ReduceMax	≥14	Cannot operate on the batch dimension.
ReduceMin	≥14	Cannot operate on the batch dimension.
ReLU	All	
ReLU6	All	
ReLUN1To1	All	
Reshape	All	Certain reshapes might not be mapped for large tensor sizes.
ResizeBilinear	All	Input/output is a 3-dimensional tensor. Depending on input/output size, this operation might not be mapped to the Edge TPU to avoid loss in precision.
ResizeNearestNeighbor	All	Input/output is a 3-dimensional tensor. Depending on input/output size, this operation might not be mapped to the Edge TPU to avoid loss in precision.
Rsqrt	≥14	
Slice	All	
Softmax	All	Supports only 1-D input tensor with a max of 16,000 elements.
SpaceToDepth	All	
Split	All	No splitting in batch dimension.
Squeeze	≤12	Supported only when input tensor dimensions that have leading 1s (that is, no relayout needed). For example input tensor with

Table 1. All operations supported by the Edge TPU and any known limitations

Operation name	Runtime version*	Known limitations
		[y][x][z] = 1,1,10 or 1,5,10 is ok. But [y][x][z] = 5,1,10 is not supported.
	≥13	None.
StridedSlice	All	Supported only when all strides are equal to 1 (that is, effectively a Stride op), and with ellipsis-axis-mask == 0, and new-axis-max == 0.
Sub	All	
Sum	≥13	Cannot operate on the batch dimension.
Squared-difference	≥14	
Tanh	All	
Transpose	≥14	
TransposeConv	≥13	

* *You must use a version of the Edge TPU Compiler that corresponds to the runtime version.*
Regardless of the operations you use, be sure you abide by the basic model requirements above.

Quantization

Quantizing your model means converting all the 32-bit floating-point numbers (such as weights and activation outputs) to the nearest 8-bit fixed-point numbers. This makes the model smaller and faster. And although these 8-bit representations can be less precise, the inference accuracy of the neural network is not significantly affected.

For compatibility with the Edge TPU, you must use either quantization-aware training (recommended) or full integer post-training quantization.
Quantization-aware training (for TensorFlow 1) uses "fake" quantization nodes in the neural network graph to simulate the effect of 8-bit values during training. Thus, this technique requires modification to the network before initial training. This generally results in a higher accuracy model (compared to post-training quantization) because it makes the model more tolerant of lower precision values, due to the fact that the 8-bit weights are learned through training rather than being converted later. It's also currently compatible with more operations than post-training quantization.
Note: As of August, 2021, the quantization-aware training API with TF2 is not compatible with the object detection API; it is compatible with image classification models only. For more compatibility, including with the object detection API, you may use quantization-aware training with TF1 or use post-training quantization with TF2.
Full integer post-training quantization doesn't require any modifications to the network, so you can use this technique to convert a previously-trained network into a quantized model. However, this conversion process requires that you supply a representative dataset. That is, you need a dataset that's formatted the same as the original training dataset (uses the same data range) and is of a similar style (it does not need to contain all the same classes, though you may use previous training/evaluation data). This representative dataset allows the

quantization process to measure the dynamic range of activations and inputs, which is critical to finding an accurate 8-bit representation of each weight and activation value.
However, not all TensorFlow Lite operations are currently implemented with an integer-only specification (they cannot be quantized using post-training quantization). By default, the TensorFlow Lite converter leaves those operations in their float format, which is not compatible with the Edge TPU. As described below, the Edge TPU Compiler stops compiling when it encounters an incompatible operation (such as a non-quantized op), and the remainder of the model executes on the CPU. So to enforce integer-only quantization, you can instruct the converter to throw an error if it encounters a non-quantizable operation. Read more about integer-only quantization.
For examples of each quantization strategy, see our Google Colab tutorials for model training.
For more details about how quantization works, read the TensorFlow Lite 8-bit quantization spec.

Float input and output tensors

As mentioned in the model requirements, the Edge TPU requires 8-bit quantized input tensors. However, if you pass the Edge TPU Compiler a model that's internally quantized but still uses float inputs, the compiler leaves a quantize op at the beginning of your graph (which runs on the CPU). Likewise, the output is dequantized at the end.

So it's okay if your TensorFlow Lite model uses float inputs/outputs. But beware that if your model uses float input and output, then there will be some amount of latency added due to the data format conversion, though it should be negligible for most models (the bigger the input tensor, the more latency you'll see).
To achieve the best performance possible, we recommend fully quantizing your model so the input and output use int8 or uint8 data, which you can do by setting the input and output type with the TF Lite converter, as shown in the TensorFlow docs for integer-only quantization.

Compiling

After you train and convert your model to TensorFlow Lite (with quantization), the final step is to compile it with the Edge TPU Compiler.
If your model does not meet all the requirements listed at the top of this section, it can still compile, but only a portion of the model will execute on the Edge TPU. At the first point in the model graph where an unsupported operation occurs, the compiler partitions the graph into two parts. The first part of the graph that contains only supported operations is compiled into a custom operation that executes on the Edge TPU, and everything else executes on the CPU, as illustrated in figure 2.
Note: Currently, the Edge TPU compiler cannot partition the model more than once, so as soon as an unsupported operation occurs, that operation and everything after it executes on the CPU, even if supported operations occur later.

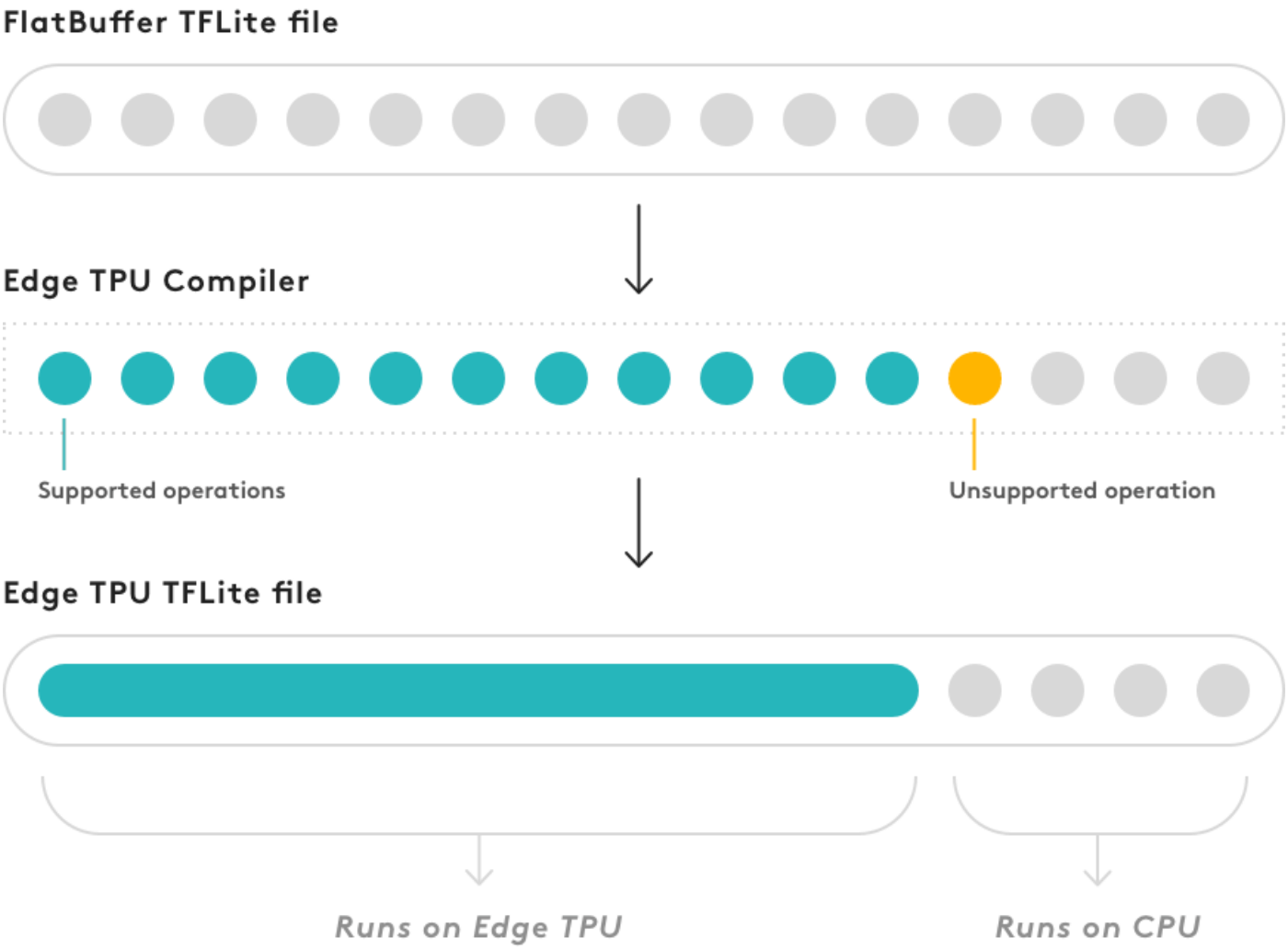


Figure 2. The compiler creates a single custom op for all Edge TPU compatible ops, until it encounters an unsupported op; the rest stays the same and runs on the CPU

If you inspect your compiled model (with a tool such as `visualize.py`), you'll see that it's still a TensorFlow Lite model except it now has a custom operation at the beginning of the graph. This custom operation is the only part of your model that is actually compiled—it contains all the operations that run on the Edge TPU. The rest of the graph (beginning with the first unsupported operation) remains the same and runs on the CPU.

If part of your model executes on the CPU, you should expect a significantly degraded inference speed compared to a model that executes entirely on the Edge TPU. We cannot predict how much slower your model will perform in this situation, so you should experiment with different architectures and strive to create a model that is 100% compatible with the Edge TPU. That is, your compiled model should contain only the Edge TPU custom operation.

Note: When compilation completes, the Edge TPU compiler tells you how many operations can execute on the Edge TPU and how many must instead execute on the CPU (if any at all). But beware that the percentage of operations that execute on the Edge TPU versus the CPU does not correspond to the overall performance impact—if even a small fraction of your model executes on the CPU, it can potentially slow the inference speed by an order of magnitude (compared to a version of the model that runs entirely on the Edge TPU).

Edge TPU inferencing overview

All inferencing with the Edge TPU is based on the TensorFlow Lite APIs (Python or C/C++). So if you already know how to run an inference using TensorFlow Lite, then running your model on the Edge TPU requires only a few new lines of code.
Note: The Edge TPU is compatible with TensorFlow Lite models only. For details about how to create a model that's compatible with the Edge TPU, read TensorFlow models on the Edge TPU.

With a compatible model in-hand, you can perform inferencing on the Edge TPU using either Python or C/C++. In either case, you also have the option to use our Coral APIs, which provide convenience functions that wrap the TensorFlow APIs and other advanced features.

Each option and the software required is described below and illustrated in figure 1.

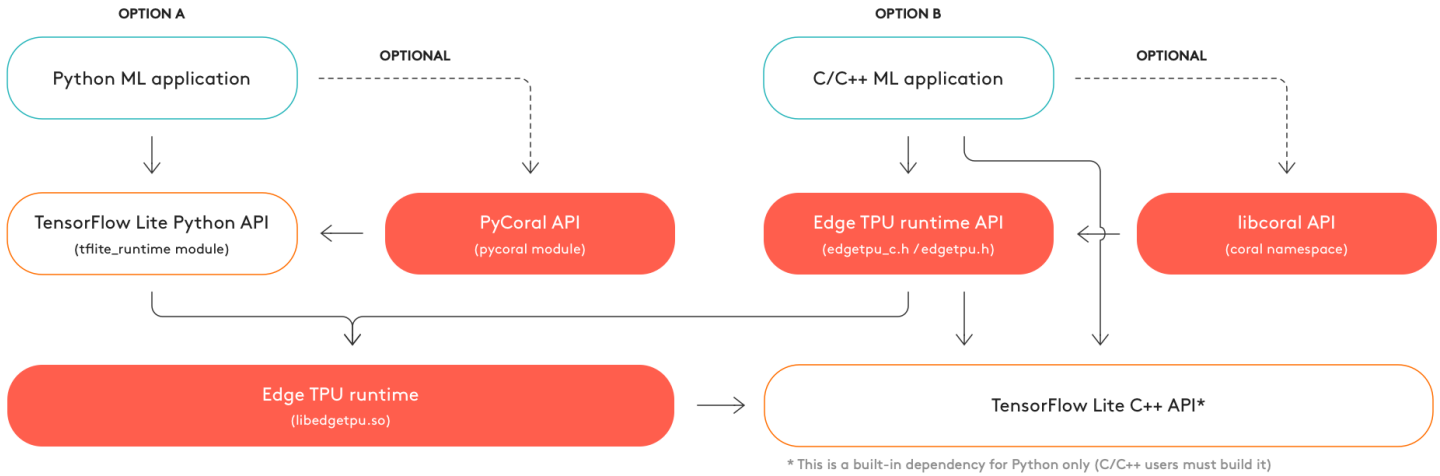


Figure 1. The three options for inferencing and the corresponding software dependencies

A. **Use the TensorFlow Lite Python API:**

   This is the standard Python API for running TensorFlow Lite models. With just a few lines of code, you can make existing TensorFlow Lite code run on the Edge TPU.

   All you need is the TensorFlow Lite Python API and the Edge TPU Runtime (libedgetpu).
   Optional: You can also use our PyCoral APIs, which provide several convenience functions for pre-processing your tensor inputs and post-processing tensor outputs for common models, and additional features such as pipelining a model with multiple Edge TPUs and on-device transfer learning.
   To get started, read Run inference on the Edge TPU with Python.

B. **Use the C/C++ TensorFlow Lite APIs:**

   This uses the standard C/C++ API for running TensorFlow Lite models. With just a few lines of code, you can make existing TensorFlow Lite code run on the Edge TPU.

   For this, you need the Edge TPU Runtime (libedgetpu)—linked statically or dynamically—and the compiled TensorFlow Lite C++ library.
   Optional: You can also use our libcoral APIs, which provide several convenience functions for pre-processing your tensor inputs and post-processing tensor outputs for common models, and additional features such as pipelining a model with multiple Edge TPUs.
   For details, read Run inference on the Edge TPU with C++.

You'll get all the libraries you need when you set up your Coral device (except for C++, which requires your own build configurations). But if you're not using one of our supported platforms, everything shown above is open sourced, so you can build these libraries yourself for your platform.

Run multiple models with multiple Edge TPUs

The Edge TPU includes a small amount of RAM that's used to store the model's parameter data locally, enabling faster inference speed compared to fetching the data from external memory. Typically, this means performance is best when running just one model per Edge TPU, because running a second model requires swapping the model's parameter data in RAM, which slows down the entire pipeline. One solution is to simply run each model on a different Edge TPU, as described on this page.
Alternatively, you might reduce the overhead cost of swapping parameter data by co-compiling your models. Co-compiling allows the Edge TPU to store the parameter data for multiple models in RAM together, which means it typically works well only for small models. To learn more about this option, read about parameter data caching and co-compiling. Otherwise, keep reading here if you want to distribute multiple models across multiple Edge TPUs.

Performance considerations

Before you add more Edge TPUs in your system, consider the following possible performance issues:

- Python does not support real multi-threading for CPU-bounded operations (read about the Python global interpreter lock (GIL)). However, we have optimized the Edge TPU Python API (but not TensorFlow Lite Python API) to work within Python's multi-threading environment for all Edge TPU operations—they are IO-bounded, which can provide performance improvements. But beware that CPU-bounded operations such as image downscaling will probably encounter a performance impact when you run multiple models because these operations cannot be multi-threaded in Python.
- When using multiple USB Accelerators, your inference speed will eventually be bottlenecked by the host USB bus's speed, especially when running large models.
- If you connect multiple USB Accelerators through a USB hub, be sure that each USB port can provide at least 500mA when using the reduced operating frequency or 900mA when using the maximum frequency (refer to the USB Accelerator performance settings). Otherwise, the device might not be able to draw enough power to function properly.
- If you use an external USB hub, connect the USB Accelerator to the primary ports only. Some USB hubs include sub-hubs with secondary ports that are not compatible—our API cannot establish an Edge TPU context on these ports. For example, if you type `lsusb -t`, you should see ports printed as shown below. The first 2 USB ports (`usbfs`) will work fine but the last one will not.

```
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/7p, 5000M

    |__ Port 3: Dev 36, If 0, Class=Hub, Driver=hub/4p, 5000M
        |__ Port 1: Dev 51, If 0, Class=Vendor Specific Class, Driver=usbfs, 5000M   # WORKS

            |__ Port 2: Dev 40, If 0, Class=Hub, Driver=hub/4p, 5000M
                |__ Port 1: Dev 41, If 0, Class=Vendor Specific Class, Driver=usbfs, 5000M   # WORKS
                    |__ Port 2: Dev 39, If 0, Class=Vendor Specific Class, Driver=usbfs, 5000M   # DOESN'T WORK
```

Using the PyCoral API

If you have multiple Edge TPUs and you want to run a specific model on each one, you must specify which device to use with each model. When using the PyCoral Python API, you just need to specify the `device` argument in `make_interpreter()`.
Caution: If you don't specify an Edge TPU when multiple are available, the same Edge TPU will probably be used for all models, which seriously affects your performance, as described in the introduction.

The `device` argument takes a string to indicate the device index position or the device type (USB or PCIe), or a combination of both. For example, this is how you can ensure each `Interpreter` is using a different Edge TPU (regardless of type):

```
# Use the first enumerated Edge TPU
interpreter_1 = make_interpreter(model_1_path, device=':0')
# Use the second enumerated Edge TPU
interpreter_2 = make_interpreter(model_2_path, device=':1')
```

Or if you want to specify USB vs PCIe device types, you can do the following:

```
# Use the first USB-based Edge TPU
interpreter_usb1 = make_interpreter(model_1_path, device='usb:0')
# Use the first PCIe-based Edge TPU
interpreter_pcie1 = make_interpreter(model_2_path, device='pci:0')
```

For more details, see the `make_interpreter()` documentation. Also check out the `two_models_inference.py` example.

Using the TensorFlow Lite Python API

If you have multiple Edge TPUs and you want to run a specific model on each one, you must specify which device to use with each model. When using the TensorFlow Lite Python API, you can do so with the `options` argument in `load_delegate()`.

Caution: If you don't specify an Edge TPU when multiple are available, the same Edge TPU will probably be used for all models, which seriously affects your performance, as described in the introduction.

The `options` argument takes a dictionary and you need just one entry, `"device"`, to specify the Edge TPU you want to use. Accepted values are the following:

- `":<N>"` : Use N-th Edge TPU
- `"usb"` : Use any USB Edge TPU
- `"usb:<N>"` : Use N-th USB Edge TPU
- `"pci"` : Use any PCIe Edge TPU
- `"pci:<N>"` : Use N-th PCIe Edge TPU

For example, this is how you can ensure each `Interpreter` is using a different Edge TPU (regardless of type):

```
# Use the first enumerated Edge TPU
interpreter_1 = Interpreter(model_1_path,
  experimental_delegates=[load_delegate('libedgetpu.so.1', options={"device": ":0"})])
# Use the second enumerated Edge TPU
interpreter_2 = Interpreter(model_2_path,
  experimental_delegates=[load_delegate('libedgetpu.so.1', options={"device": ":1"})])
```

Or if you want to specify USB vs PCIe device types, you can do the following:

```
# Use the first USB-based Edge TPU
interpreter_usb1 = Interpreter(model_1_path,
  experimental_delegates=[load_delegate('libedgetpu.so.1', options={"device": "usb:0"})])
# Use the first PCIe-based Edge TPU
interpreter_pci2 = Interpreter(model_2_path,
  experimental_delegates=[load_delegate('libedgetpu.so.1', options={"device": "pci:0"})])
```

> **Note:** If you're not running Linux, your delegate filename (`libedgetpu.so.1`) will be different (see how to add the delegate).

Using the TensorFlow Lite C++ API

If you're using the TensorFlow Lite C++ API to run inference and you have multiple Edge TPUs, you can specify which Edge TPU each `Interpreter` should use when you create the `EdgeTpuContext` via `EdgeTpuManager::OpenDevice()`.

Caution: If you don't specify an Edge TPU when multiple are available, the same Edge TPU will probably be used for all models, which seriously affects your performance, as described in the introduction.
The `OpenDevice()` method includes a parameter for `device_type`, which accepts one of two values:

- `DeviceType.kApexUsb`: Use the default USB-connected Edge TPU.
- `DeviceType.kApexPci`: Use the default PCIe-connected Edge TPU.

If you have multiple Edge TPUs of the same type, then you must specify the second parameter, `device_path`. To get the specific device path for each available Edge TPU, call `EdgeTpuManager.EnumerateEdgeTpu()`.
For an example, see `two_models_two_tpus_threaded.cc`.

Using the Edge TPU Python API (deprecated)

The Edge TPU Python API is deprecated. Instead try the PyCoral API.
If you're using the Edge TPU Python API to run inference and you have multiple Edge TPUs, the Edge TPU API automatically assigns each inference engine (such as `ClassificationEngine` and `DetectionEngine`) to a different Edge TPU. So you don't need to write any extra code if you have an equal number of inference engines and Edge TPUs—unlike the TensorFlow Lite API above.

For example, if you have two Edge TPUs and two models, you can run each model on separate Edge TPUs by simply creating the inference engines as usual:

```
# Each engine is automatically assigned to a different Edge TPU
engine_a = ClassificationEngine(classification_model)
engine_b = DetectionEngine(detection_model)
```

Then they'll automatically run on separate Edge TPUs.

If you have just one Edge TPU, then this code still works and they both use the same Edge TPU.
However, if you have multiple Edge TPUs (*N*) and you have *N + 1* (or more) models, then you must specify which Edge TPU to use for each additional inference engine. Otherwise, you'll receive an error that says your engine does not map to an Edge TPU device.
For example, if you have two Edge TPUs and three models, you must set the third engine to run on the same Edge TPU as one of the others (you decide which). The following code shows how you can do this for `engine_c` by specifying the `device_path` argument to be the same device used by `engine_b`:

```
# The second engine is purposely assigned to the same Edge TPU as the first
engine_a = ClassificationEngine(classification_model)
engine_b = DetectionEngine(detection_model)
engine_c = DetectionEngine(other_detection_model, engine_b.device_path())
```

For example code, see `two_models_inference.py`.
Note: All Edge TPUs connected over USB are treated equally; there's no prioritization when distributing the models. But if you attach a USB Accelerator to a Dev Board, the system always prefers the on-board (PCIe) Edge TPU before using the USB devices.