Compute Module hardware

Datasheets and Schematics

Edit this on GitHub

Compute Module 4

The latest version of the Compute Module is the Compute Module 4 (CM4). It is the recommended Compute Module for all current and future development.

- Compute Module 4 Datasheet
- Compute Module 4 IO Board Datasheet

NOTE

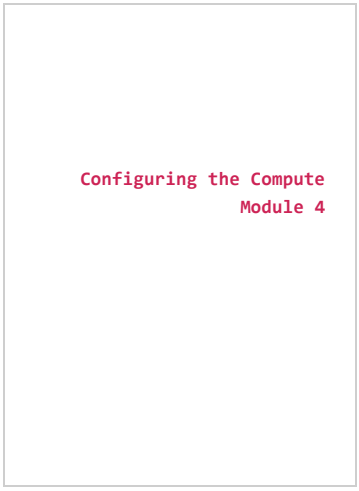
> Schematics are not available for the Compute Module 4, but are available for the IO board. Schematics for the CMIO4 board are included in the datasheet.

There is also a KiCad PCB design set available:

- Compute Module 4 IO Board KiCad files

WHITEPAPER

Configuring the Compute Module 4



The Raspberry Pi Compute Module 4 (CM 4) is available in a number of different hardware configurations. Sometimes it may be necessary to disable some of these features when they are not required.

This document describes how to disable various hardware interfaces, in both hardware and software, and how to reduce the amount of memory used by the Linux operating system (OS).

Older Products

Raspberry Pi CM1, CM3 and CM3L are supported products with an End-of-Life (EOL) date no earlier than January 2026. The Compute Module 3+ offers improved thermal performance, and a wider range of Flash memory options.

- Compute Module 1 and Compute Module 3

Raspberry Pi CM3+ and CM3+ Lite are supported prodicts with an End-of-Life (EOL) date no earlier than January 2026.

- Compute Module 3+

Schematics for the Compute Module 1, 3 and 3L

- CM1 Rev 1.1
- CM3 and CM3L Rev 1.0

Schematics for the Compute Module IO board (CMIO):

- CMIO Rev 3.0 (Supports CM1, CM3, CM3L, CM3+ and CM3+L)

Schematics for the Compute Module camera/display adapter board (CMCDA):

- CMCDA Rev 1.1

WHITEPAPER

Transitioning from CM3 to CM4

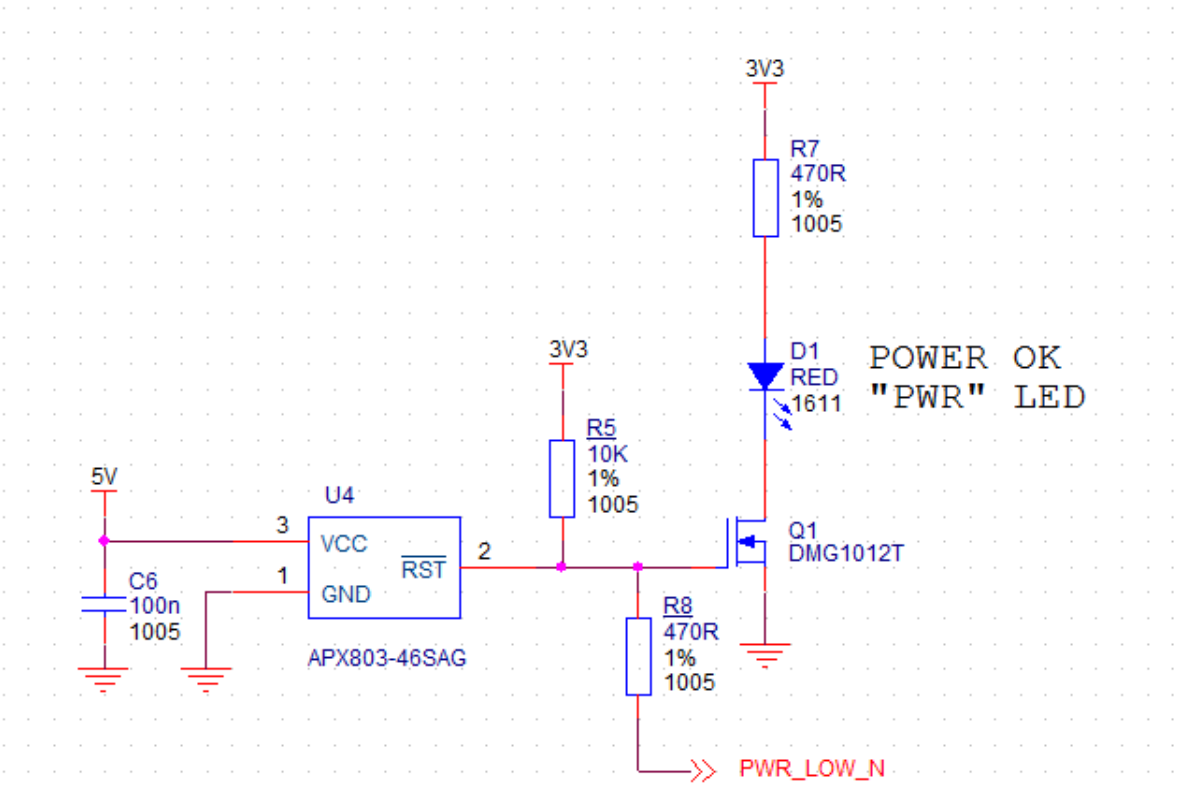
Transitioning from CM3 to CM4

This whitepaper is for those who wish to move from using a Raspberry Pi Compute Module (CM) 1 or 3 to a Raspberry Pi CM 4.

From a software perspective, the move from Raspberry Pi CM 1/3 to Raspberry Pi CM 4 is relatively painless, as Raspberry Pi OS should work on all platforms.

Under Voltage Detection

Schematic for an under-voltage detection circuit, as used in older models of Raspberry Pi:



Design Files for CMIO Boards

Edit this on GitHub

Compute Module IO board for CM4

Design data for the Compute Module 4 IO board can be found in its datasheet:

- Compute Module 4 IO Board datasheet

There is also a KiCad PCB design set available:

- Compute Module 4 IO Board KiCad files

Older Products

- CMIO Rev 1.2
- CMIO Rev 3.0

Design data for the Compute Module camera/display adapter board (CMCDA):

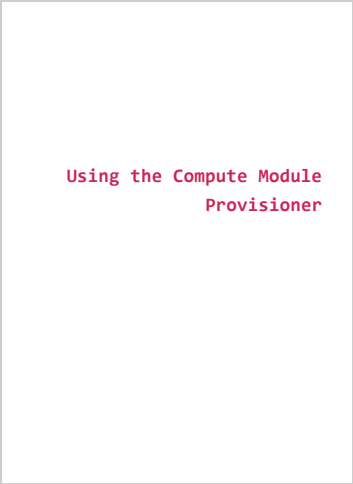
- CMCDA Rev 1.1

Flashing the Compute Module eMMC

Edit this on GitHub

WHITEPAPER

Using the Compute Module Provisioner



The CM Provisioner is a web application designed to make programming a large number of Raspberry Pi Compute Module (CM) devices much easier and quicker. It is simple to install and simple to use.

It provides an interface to a database of kernel images that can be uploaded, along with the ability to use scripts to customise various parts of the installation during the flashing process. Label printing and firmware updating is also supported.

The Compute Module has an on-board eMMC device connected to the primary SD card interface. This guide explains how to write data to the eMMC storage using a Compute Module IO board.

Please also read the section in the Compute Module Datasheets

IMPORTANT

For mass provisioning of CM3, CM3+ and CM4 the Raspberry Pi Compute Module Provisioning System is recommended.

Steps to Flash the eMMC

To flash the Compute Module eMMC, you either need a Linux system (a Raspberry Pi is recommended, or Ubuntu on a PC) or a Windows system (Windows 10 is recommended). For BCM2837 (CM3), a bug which affected the Mac has been fixed, so this will also work.

NOTE

There is a bug in the BCM2835 (CM1) bootloader which returns a slightly incorrect USB packet to the host. Most USB hosts seem to ignore this benign bug and work fine; we do, however, see some USB ports that don't work due to this bug. We don't quite understand why some ports fail, as it doesn't seem to be correlated with whether they are USB2 or USB3 (we have seen both types working), but it's likely to be specific to the host controller and driver. This bug has been fixed in BCM2837.

Setting up the CMIO board

Compute Module 4

Ensure the Compute Module is fitted correctly installed on the IO board. It should lie flat on the IO board.

- Make sure that `nRPI_BOOT` which is on J2 (`disable eMMC Boot`) on the IO board jumper is fitted
- Use a micro USB cable to connect the micro USB slave port J11 on IO board to the host device.
- Do not power up yet.

Compute Module 1 and 3

Ensure the Compute Module itself is correctly installed on the IO board. It should lie parallel with the board, with the engagement clips clicked into place.

- Make sure that J4 (USB SLAVE BOOT ENABLE) is set to the 'EN' position.
- Use a micro USB cable to connect the micro USB slave port J15 on IO board to the host device.
- Do not power up yet.

For Windows Users

Under Windows, an installer is available to install the required drivers and boot tool automatically. Alternatively, a user can compile and run it using Cygwin and/or install the drivers manually.

Windows Installer

For those who just want to enable the Compute Module eMMC as a mass storage device under Windows, the stand-alone installer is the recommended option. This installer has been tested on Windows 10 64-bit.

Please ensure you are not writing to any USB devices whilst the installer is running.

1. Download and run the Windows installer to install the drivers and boot tool.
2. Plug your host PC USB into the USB SLAVE port, making sure you have setup the board as described above.
3. Apply power to the board; Windows should now find the hardware and install the driver.
4. Once the driver installation is complete, run the `RPiBoot.exe` tool that was previously installed.
5. After a few seconds, the Compute Module eMMC will pop up under Windows as a disk (USB mass storage device).

Building `rpiboot` on your host system.

Instructions for building and running the latest release of `rpiboot` are documented in the usbboot readme on Github.

Writing to the eMMC (Windows)

After `rpiboot` completes, a new USB mass storage drive will appear in Windows. We

recommend using Raspberry Pi Imager to write images to the drive.

Make sure J4 (USB SLAVE BOOT ENABLE) / J2 (nRPI_BOOT) is set to the disabled position and/or nothing is plugged into the USB slave port. Power cycling the IO board should now result in the Compute Module booting from eMMC.

Writing to the eMMC (Linux)

After `rpiboot` completes, you will see a new device appear; this is commonly `/dev/sda` on a Raspberry Pi but it could be another location such as `/dev/sdb`, so check in `/dev/` or run `lsblk` before running `rpiboot` so you can see what changes.

You now need to write a raw OS image (such as Raspberry Pi OS) to the device. Note the following command may take some time to complete, depending on the size of the image: (Change `/dev/sdX` to the appropriate device.)

```
sudo dd if=raw_os_image_of_your_choice.img of=/dev/sdX bs=4MiB
```

Once the image has been written, unplug and re-plug the USB; you should see two partitions appear (for Raspberry Pi OS) in `/dev`. In total, you should see something similar to this:

```
/dev/sdX    <- Device
/dev/sdX1   <- First partition (FAT)
/dev/sdX2   <- Second partition (Linux filesystem)
```

The `/dev/sdX1` and `/dev/sdX2` partitions can now be mounted normally.

Make sure J4 (USB SLAVE BOOT ENABLE) / J2 (nRPI_BOOT) is set to the disabled position and/or nothing is plugged into the USB slave port. Power cycling the IO board should now result in the Compute Module booting from eMMC.

Compute Module 4 Bootloader

The default bootloader configuration on CM4 is designed to support bringup and development on a Compute Module 4 IO board and the software version flashed at manufacture may be older than the latest release. For final products please consider:

- Selecting and verifying a specific bootloader release. The version in the `usbboot` repo is always a recent stable release.

- Configuring the boot device (e.g. network boot). See `BOOT_ORDER` section in the bootloader configuration guide.
- Enabling hardware write-protection on the bootloader EEPROM to ensure that the bootloader can't be modified on inaccessible products (such as remote or embedded devices).

NOTE

The Compute Module 4 ROM never runs `recovery.bin` from SD/EMMC and the `rpi-eeprom-update` service is not enabled by default. This is necessary because the EMMC is not removable and an invalid `recovery.bin` file would prevent the system from booting. This can be overridden and used with `self-update` mode where the bootloader can be updated from USB MSD or Network boot. However, `self-update` mode is not an atomic update and therefore not safe in the event of a power failure whilst the EEPROM was being updated.

Flashing storage devices other than SD cards

The new Linux-based mass-storage gadget supports flashing of NVMe, EMMC and USB block devices.

This is normally faster than using the `rpiboot` firmware driver and also provides a UART console to the device for easier debug.

See also: CM4 rpiboot extensions

Modifying the bootloader configuration

To modify the CM4 bootloader configuration:

- Navigate to the `usbboot/recovery` directory
- Replace `pieeprom.original.bin` if a specific bootloader release is required.
- Edit the default `boot.conf` bootloader configuration file. Typically, at least the BOOT_ORDER must be updated:
  - For network boot, use `BOOT_ORDER=0xf2`
  - For SD/EMMC boot, use `BOOT_ORDER=0xf1`
  - For USB boot failing over to EMMC, use `BOOT_ORDER=0xf15`
- Run `./update-pieeprom.sh` to update the EEPROM image `pieeprom.bin` image file.

- If EEPROM write-protection is required, edit `config.txt` and add `eeprom_write_protect=1`. Hardware write-protection must be enabled via software and then locked by pulling the `EEPROM_nWP` pin low.
- Run `../rpiboot -d .` to update the bootloader using the updated EEPROM image `pieeprom.bin`

The `pieeprom.bin` file is now ready to be flashed to the Compute Module 4.

Flashing the bootloader EEPROM - Compute Module 4

To flash the bootloader EEPROM follow the same hardware setup as for flashing the EMMC but also ensure EEPROM_nWP is NOT pulled low. Once complete `EEPROM_nWP` may be pulled low again.

```
# Writes recovery/pieeprom.bin to the bootloader EEPROM.
./rpiboot -d recovery
```

Troubleshooting

For a small percentage of Raspberry Pi Compute Module 3s, booting problems have been reported. We have traced these back to the method used to create the FAT32 partition; we believe the problem is due to a difference in timing between the BCM2835/6/7 and the newer eMMC devices. The following method of creating the partition is a reliable solution in our hands.

```
sudo parted /dev/<device>
(parted) mkpart primary fat32 4MiB 64MiB
(parted) q
sudo mkfs.vfat -F32 /dev/<device>
sudo cp -r <files>/* <mountpoint>
```

Attaching and Enabling Peripherals

Edit this on GitHub

NOTE

Unless explicitly stated otherwise, these instructions will work identically on Compute Module 1 and Compute Module 3 and their CMIO board(s).

This guide is designed to help developers using the Compute Module 1 (and Compute Module 3) get to grips with how to wire up peripherals to the Compute Module pins, and how to make changes to the software to enable these peripherals to work correctly.

The Compute Module 1 (CM1) and Compute Module 3 (CM3) contain the Raspberry Pi BCM2835 (or BCM2837 for CM3) system on a chip (SoC) or 'processor', memory, and eMMC. The eMMC is similar to an SD card but is soldered onto the board. Unlike SD cards, the eMMC is specifically designed to be used as a disk and has extra features that make it more reliable in this use case. Most of the pins of the SoC (GPIO, two CSI camera interfaces, two DSI display interfaces, HDMI etc) are freely available and can be wired up as the user sees fit (or, if unused, can usually be left unconnected). The Compute Module is a DDR2 SODIMM form-factor-compatible module, so any DDR2 SODIMM socket should be able to be used

NOTE

> The pinout is NOT the same as an actual SODIMM memory module.

To use the Compute Module, a user needs to design a (relatively simple) 'motherboard' which can provide power to the Compute Module (3.3V and 1.8V at minimum), and which connects the pins to the required peripherals for the user's application.

Raspberry Pi provides a minimal motherboard for the Compute Module (called the Compute Module IO Board, or CMIO Board) which powers the module, brings out the GPIO to pin headers, and brings the camera and display interfaces out to FFC connectors. It also provides HDMI, USB, and an 'ACT' LED, as well as the ability to program the eMMC of a module via USB from a PC or Raspberry Pi.

This guide first explains the boot process and how Device Tree is used to describe attached hardware; these are essential things to understand when designing with the Compute Module. It then provides a worked example of attaching an I2C and an SPI peripheral to a CMIO (or CMIO V3 for CM3) Board and creating the Device Tree files necessary to make both peripherals work under Linux, starting from a vanilla Raspberry Pi OS image.

BCM283x GPIOs

BCM283x has three banks of General-Purpose Input/Output (GPIO) pins: 28 pins on Bank 0, 18 pins on Bank 1, and 8 pins on Bank 2, making 54 pins in total. These pins can be used as true GPIO pins, i.e. software can set them as inputs or outputs, read and/or set state, and use them as interrupts. They also can be set to 'alternate functions' such as I2C, SPI, I2S, UART, SD card, and others.

On a Compute Module, both Bank 0 and Bank 1 are free to use. Bank 2 is used for eMMC and HDMI hot plug detect and ACT LED / USB boot control.

It is useful on a running system to look at the state of each of the GPIO pins (what function they are set to, and the voltage level at the pin) so that you can see if the system is set up as expected. This is particularly helpful if you want to see if a Device Tree is working as expected, or to get a look at the pin states during hardware debug.

Raspberry Pi provides the `pinctrl` package which is a tool for hacking and debugging GPIO.

BCM283x Boot Process

BCM283x devices consist of a VideoCore GPU and ARM CPU cores. The GPU is in fact a system consisting of a DSP processor and hardware accelerators for imaging, video encode and decode, 3D graphics, and image compositing.

In BCM283x devices, it is the DSP core in the GPU that boots first. It is responsible for general setup and housekeeping before booting up the main ARM processor(s).

The BCM283x devices as used on Raspberry Pi and Compute Module boards have a three-stage boot process:

1. The GPU DSP comes out of reset and executes code from a small internal ROM (the boot ROM). The sole purpose of this code is to load a second stage boot loader via one of the external interfaces. On a Raspberry Pi or Compute Module, this code first looks for a second stage boot loader on the SD card (eMMC); it expects this to be called `bootcode.bin` and to be on the first partition (which must be FAT32). If no SD card is found or `bootcode.bin` is not found, the Boot ROM sits and waits in 'USB boot' mode, waiting for a host to give it a second stage boot loader via the USB interface.

2. The second stage boot loader (`bootcode.bin` on the sdcard or `usbbootcode.bin` for usb boot) is responsible for setting up the LPDDR2 SDRAM interface and various other critical system functions and then loading and executing the main GPU firmware (called `start.elf`, again on the primary SD card partition).

3. `start.elf` takes over and is responsible for further system setup and booting up the ARM processor subsystem, and contains the firmware that runs on the various parts of the GPU. It first reads `dt-blob.bin` to determine initial GPIO pin states and GPU-specific interfaces and clocks, then parses `config.txt`. It then loads an ARM device tree file (e.g. `bcm2708-rpi-cm.dtb` for a Compute Module 1) and any device tree overlays specified in `config.txt` before starting the ARM subsystem and passing

the device tree data to the booting Linux kernel.

Device Tree

Device Tree is a special way of encoding all the information about the hardware attached to a system (and consequently required drivers).

On a Raspberry Pi or Compute Module there are several files in the first FAT partition of the SD/eMMC that are binary 'Device Tree' files. These binary files (usually with extension `.dtb`) are compiled from human-readable text descriptions (usually files with extension `.dts`) by the Device Tree compiler.

On a standard Raspberry Pi OS image in the first (FAT) partition you will find two different types of device tree files, one is used by the GPU only and the rest are standard ARM device tree files for each of the BCM283x based Raspberry Pi products:

- `dt-blob.bin` (used by the GPU)
- `bcm2708-rpi-b.dtb` (Used for Raspberry Pi 1 Models A and B)
- `bcm2708-rpi-b-plus.dtb` (Used for Raspberry Pi 1 Models B+ and A+)
- `bcm2709-rpi-2-b.dtb` (Used for Raspberry Pi 2 Model B)
- `bcm2710-rpi-3-b.dtb` (Used for Raspberry Pi 3 Model B)
- `bcm2708-rpi-cm.dtb` (Used for Raspberry Pi Compute Module 1)
- `bcm2710-rpi-cm3.dtb` (Used for Raspberry Pi Compute Module 3)

NOTE

`dt-blob.bin` by default does not exist as there is a 'default' version compiled into `start.elf`, but for Compute Module projects it will often be necessary to provide a `dt-blob.bin` (which overrides the default built-in file).

NOTE

`dt-blob.bin` is in compiled device tree format, but is only read by the GPU firmware to set up functions exclusive to the GPU - see below.

- A guide to creating `dt-blob.bin`.
- A guide to the Linux Device Tree for Raspberry Pi.

During boot, the user can specify a specific ARM device tree to use via the `device_tree` parameter in `config.txt`, for example adding the line `device_tree=mydt.dtb` to `config.txt` where `mydt.dtb` is the dtb file to load instead of one of the standard ARM dtb files. While a user can create a full device tree for their Compute Module product, the recommended way to add hardware is to use overlays (see next section).

In addition to loading an ARM dtb, `start.elf` supports loading additional Device Tree 'overlays' via the `dtoverlay` parameter in `config.txt`, for example adding as many `dtoverlay=myoverlay` lines as required as overlays to `config.txt`, noting that overlays live in `/overlays` and are suffixed `-overlay.dtb` e.g. `/overlays/myoverlay-overlay.dtb`. Overlays are merged with the base dtb file before the data is passed to the Linux kernel when it starts.

Overlays are used to add data to the base dtb that (nominally) describes non-board-specific hardware. This includes GPIO pins used and their function, as well as the device(s) attached, so that the correct drivers can be loaded. The convention is that on a Raspberry Pi, all hardware attached to the Bank0 GPIOs (the GPIO header) should be described using an overlay. On a Compute Module all hardware attached to the Bank0 and Bank1 GPIOs should be described in an overlay file. You don't have to follow these conventions: you can roll all the information into one single dtb file, as previously described, replacing `bcm2708-rpi-cm.dtb`. However, following the conventions means that you can use a 'standard' Raspberry Pi OS release, with its standard base dtb and all the product-specific information contained in a separate overlay. Occasionally the base dtb might change - usually in a way that will not break overlays - which is why using an overlay is suggested.

dt-blob.bin

When `start.elf` runs, it first reads something called `dt-blob.bin`. This is a special form of Device Tree blob which tells the GPU how to (initially) set up the GPIO pin states, and also any information about GPIOs/peripherals that are controlled (owned) by the GPU, rather than being used via Linux on the ARM. For example, the Raspberry Pi Camera peripheral is managed by the GPU, and the GPU needs exclusive access to an I2C interface to talk to it, as well as a couple of control pins. I2C0 on most Raspberry Pi Boards and Compute Modules is nominally reserved for exclusive GPU use. The information on which GPIO pins the GPU should use for I2C0, and to control the camera functions, comes from `dt-blob.bin`.

NOTE

> The `start.elf` firmware has a 'built-in' default `dt-blob.bin` which is used if no `dt-blob.bin` is found on the root of the first FAT partition. Most Compute Module projects will want to provide their own custom `dt-blob.bin`. Note that `dt-blob.bin` specifies

which pin is for HDMI hot plug detect, although this should never change on Compute Module. It can also be used to set up a GPIO as a GPCLK output, and specify an ACT LED that the GPU can use while booting. Other functions may be added in future.

minimal-cm-dt-blob.dts is an example `.dts` device tree file that sets up the HDMI hot plug detect and ACT LED and sets all other GPIOs to be inputs with default pulls.

To compile the `minimal-cm-dt-blob.dts` to `dt-blob.bin` use the Device Tree Compiler `dtc`:

```
dtc -I dts -O dtb -o dt-blob.bin minimal-cm-dt-blob.dts
```

ARM Linux Device Tree

After `start.elf` has read `dt-blob.bin` and set up the initial pin states and clocks, it reads `config.txt` which contains many other options for system setup.

After reading `config.txt` another device tree file specific to the board the hardware is running on is read: this is `bcm2708-rpi-cm.dtb` for a Compute Module 1, or `bcm2710-rpi-cm.dtb` for Compute Module 3. This file is a standard ARM Linux device tree file, which details how hardware is attached to the processor: what peripheral devices exist in the SoC and where, which GPIOs are used, what functions those GPIOs have, and what physical devices are connected. This file will set up the GPIOs appropriately, overwriting the pin state set up in `dt-blob.bin` if it is different. It will also try to load driver(s) for the specific device(s).

Although the `bcm2708-rpi-cm.dtb` file can be used to load all attached devices, the recommendation for Compute Module users is to leave this file alone. Instead, use the one supplied in the standard Raspberry Pi OS software image, and add devices using a custom 'overlay' file as previously described. The `bcm2708-rpi-cm.dtb` file contains (disabled) entries for the various peripherals (I2C, SPI, I2S etc.) and no GPIO pin definitions, apart from the eMMC/SD Card peripheral which has GPIO defs and is enabled, because it is always on the same pins. The idea is that the separate overlay file will enable the required interfaces, describe the pins used, and also describe the required drivers. The `start.elf` firmware will read and merge the `bcm2708-rpi-cm.dtb` with the overlay data before giving the merged device tree to the Linux kernel as it boots up.

Device Tree Source and Compilation

The Raspberry Pi OS image provides compiled dtb files, but where are the source dts files? They live in the Raspberry Pi Linux kernel branch, on GitHub. Look in the `arch/arm/boot/`

`dts` folder.

Some default overlay dts files live in `arch/arm/boot/dts/overlays`. Corresponding overlays for standard hardware that can be attached to a **Raspberry Pi** in the Raspberry Pi OS image are on the FAT partition in the `/overlays` directory. Note that these assume certain pins on BANK0, as they are for use on a Raspberry Pi. In general, use the source of these standard overlays as a guide to creating your own, unless you are using the same GPIO pins as you would be using if the hardware was plugged into the GPIO header of a Raspberry Pi.

Compiling these dts files to dtb files requires an up-to-date version of the Device Tree compiler `dtc`. The way to install an appropriate version on Raspberry Pi is to run:

```
sudo apt install device-tree-compiler
```

If you are building your own kernel then the build host also gets a version in `scripts/dtc`. You can arrange for your overlays to be built automatically by adding them to `Makefile` in `arch/arm/boot/dts/overlays`, and using the 'dtbs' make target.

Device Tree Debugging

When the Linux kernel is booted on the ARM core(s), the GPU provides it with a fully assembled device tree, assembled from the base dts and any overlays. This full tree is available via the Linux proc interface in `/proc/device-tree`, where nodes become directories and properties become files.

You can use `dtc` to write this out as a human readable dts file for debugging. You can see the fully assembled device tree, which is often very useful:

```
dtc -I fs -O dts -o proc-dt.dts /proc/device-tree
```

As previously explained in the GPIO section, it is also very useful to use `pinctrl` to look at the setup of the GPIO pins to check that they are as you expect. If something seems to be going awry, useful information can also be found by dumping the GPU log messages:

```
sudo vclog --msg
```

You can include more diagnostics in the output by adding `dtdebug=1` to `config.txt`.

Examples

NOTE

> Please use the Device Tree subforum on the Raspberry Pi forums to ask Device Tree related questions.

For these simple examples I used a CMIO board with peripherals attached via jumper wires.

For each of the examples we assume a CM1+CMIO or CM3+CMIO3 board with a clean install of the latest Raspberry Pi OS Lite version on the Compute Module.

The examples here require internet connectivity, so a USB hub plus keyboard plus wireless LAN or Ethernet dongle plugged into the CMIO USB port is recommended.

Please post any issues, bugs or questions on the Raspberry Pi Device Tree subforum.

Example 1 - attaching an I2C RTC to BANK1 pins

In this simple example we wire an NXP PCF8523 real time clock (RTC) to the CMIO board BANK1 GPIO pins: 3V3, GND, I2C1_SDA on GPIO44 and I2C1_SCL on GPIO45.

Download minimal-cm-dt-blob.dts and copy it to the SD card FAT partition, located in `/boot/firmware/` when the Compute Module has booted.

Edit `minimal-cm-dt-blob.dts` and change the pin states of GPIO44 and 45 to be I2C1 with pull-ups:

```
sudo nano /boot/firmware/minimal-cm-dt-blob.dts
```

Change lines:

```
pin@p44 { function = "input"; termination = "pull_down"; }; // DEFAULT STATE WA
S INPUT NO PULL
pin@p45 { function = "input"; termination = "pull_down"; }; // DEFAULT STATE WA
S INPUT NO PULL
```

to:

```
pin@p44 { function = "i2c1"; termination = "pull_up"; }; // SDA1
pin@p45 { function = "i2c1"; termination = "pull_up"; }; // SCL1
```

NOTE

We could use this `dt-blob.dts` with no changes The Linux Device Tree will (re)configure these pins during Linux kernel boot when the specific drivers are loaded, so it is up to you whether you modify `dt-blob.dts`. I like to configure `dt-blob.dts` to what I expect the final GPIOs to be, as they are then set to their final state as soon as possible during the GPU boot stage, but this is not strictly necessary. You may find that in some cases you do need pins to be configured at GPU boot time, so they are in a specific state when Linux drivers are loaded. For example, a reset line may need to be held in the correct orientation.

Compile `dt-blob.bin`:

```
sudo dtc -I dts -O dtb -o /boot/firmware/dt-blob.bin /boot/firmware/minimal-cm-dt-blob.dts
```

Grab example1-overlay.dts, put it in `/boot/firmware/`, then compile it:

```
sudo dtc -@ -I dts -O dtb -o /boot/firmware/overlays/example1.dtbo /boot/firmware/example1-overlay.dts
```

NOTE

The '-@' in the `dtc` command line. This is necessary if you are compiling dts files with external references, as overlays tend to be.

Edit `/boot/firmware/config.txt` and add the line:

```
dtoverlay=example1
```

Now save and reboot.

Once rebooted, you should see an rtc0 entry in /dev. Running:

```
sudo hwclock
```

will return with the hardware clock time, and not an error.

Example 2 - Attaching an ENC28J60 SPI Ethernet Controller on BANK0

In this example we use one of the already available overlays in `/boot/firmware/overlays` to add an ENC28J60 SPI Ethernet controller to BANK0. The Ethernet controller is connected to SPI pins CE0, MISO, MOSI and SCLK (GPIO8-11 respectively), as well as GPIO25 for a falling edge interrupt, and of course GND and 3V3.

In this example we won't change `dt-blob.bin`, although of course you can if you wish. We should see that Linux Device Tree correctly sets up the pins.

Edit `/boot/firmware/config.txt` and add the following line:

```
dtoverlay=enc28j60
```

Now save and reboot.

Once rebooted you should see, as before, an rtc0 entry in /dev. Running:

```
sudo hwclock
```

will return with the hardware clock time, and not an error.

You should also have Ethernet connectivity:

```
ping 8.8.8.8
```

should work.

finally running:

```
pinctrl
```

should show that GPIO8-11 have changed to ALT0 (SPI) functions.

Attach a Raspberry Pi Camera Module

Edit this on GitHub

The Compute Module has two CSI-2 camera interfaces: CAM1 and CAM0. This section explains how to connect one or two Raspberry Pi Cameras to a Compute Module using the CAM1 and CAM0 interfaces with a Compute Module I/O Board.

Update your system

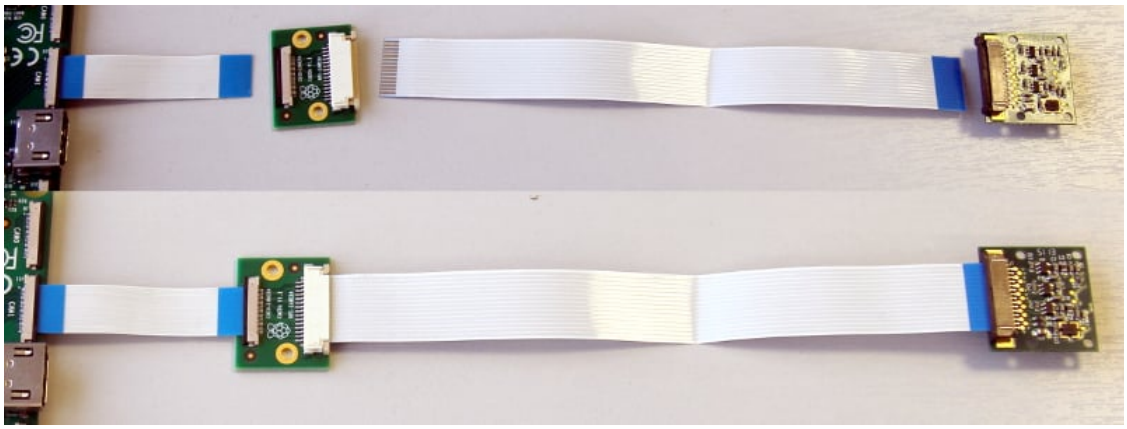
Before configuring a camera, ensure your system runs the latest available software:

```
sudo apt update
sudo apt full-upgrade
```

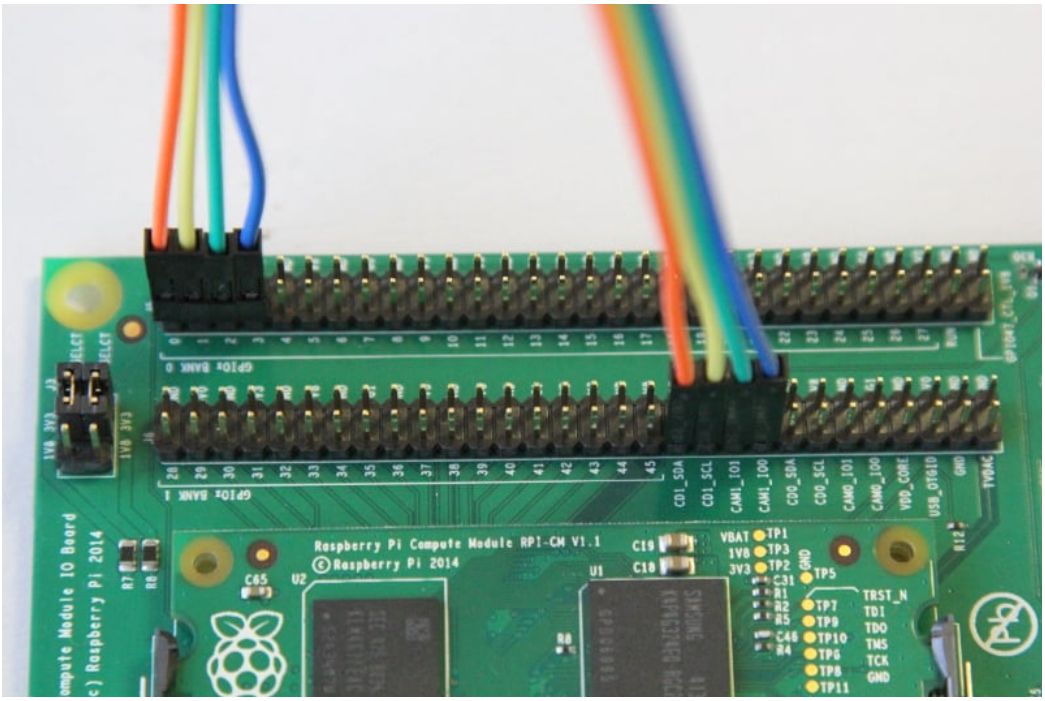
Connect one camera

To connect a single camera to a Compute Module, complete the following steps:

1. Disconnect the Compute Module from power.
2. Connect the Camera Module to the CAM1 port using a RPI-CAMERA board or a Raspberry Pi Zero camera cable.



3. *(CM1, CM3, CM3+, and CM4S only)*: Connect the following GPIO pins with jumper cables:
    - `0` to `CD1_SDA`
    - `1` to `CD1_SCL`
    - `2` to `CAM1_I01`
    - `3` to `CAM1_I00`



4. Reconnect the Compute Module to power.

5. Remove (or comment out with the prefix `#`) the following lines, if they exist, in `/boot/firmware/config.txt`:

```
camera_auto_detect=1
```

```
dtparam=i2c_arm=on
```

6. *(CM1, CM3, CM3+, and CM4S only)*: Add the following directive to `/boot/firmware/config.txt` to accommodate the swapped GPIO pin assignment on the I/O board:

```
dtoverlay=cm-swap-i2c0
```

7. *(CM1, CM3, CM3+, and CM4S only)*: Add the following directive to `/boot/firmware/config.txt` to assign GPIO 3 as the CAM1 regulator:

```
dtparam=cam1_reg
```

8. Add the appropriate directive to `/boot/firmware/config.txt` to manually configure

the driver for your camera model:

camera model	directive
v1 camera	`dtoverlay=ov5647,cam1`
v2 camera	`dtoverlay=imx219,cam1`
v3 camera	`dtoverlay=imx708,cam1`
HQ camera	`dtoverlay=imx477,cam1`
GS camera	`dtoverlay=imx296,cam1`

9. Reboot your Compute Module with `sudo reboot`.

10. Run the following command to check the list of detected cameras:

```
rpicam-hello --list
```

You should see your camera model, referred to by the driver directive in the table above, in the output.

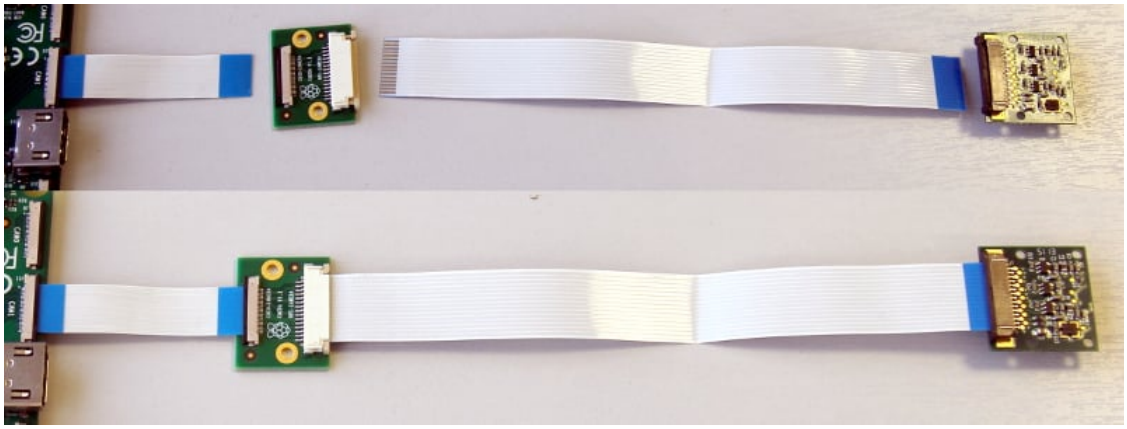
Connect two cameras

To connect two cameras to a Compute Module, complete the following steps:

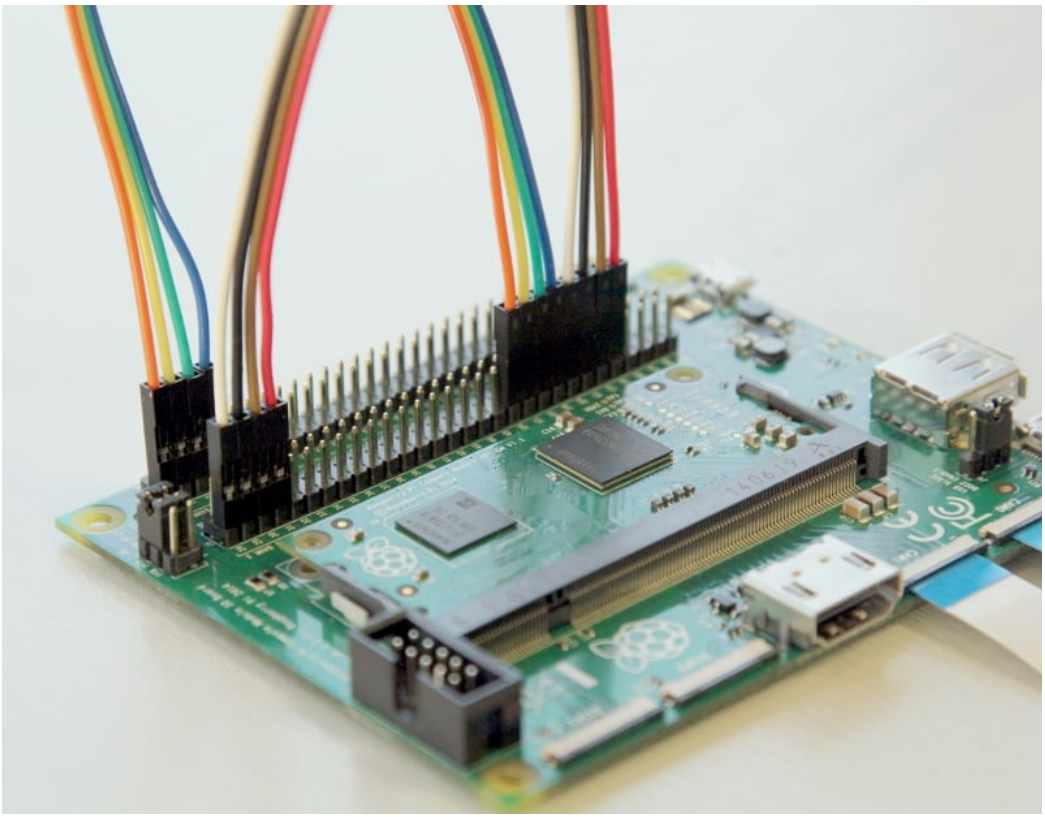
1. Follow the single camera instructions above.

2. Disconnect the Compute Module from power.

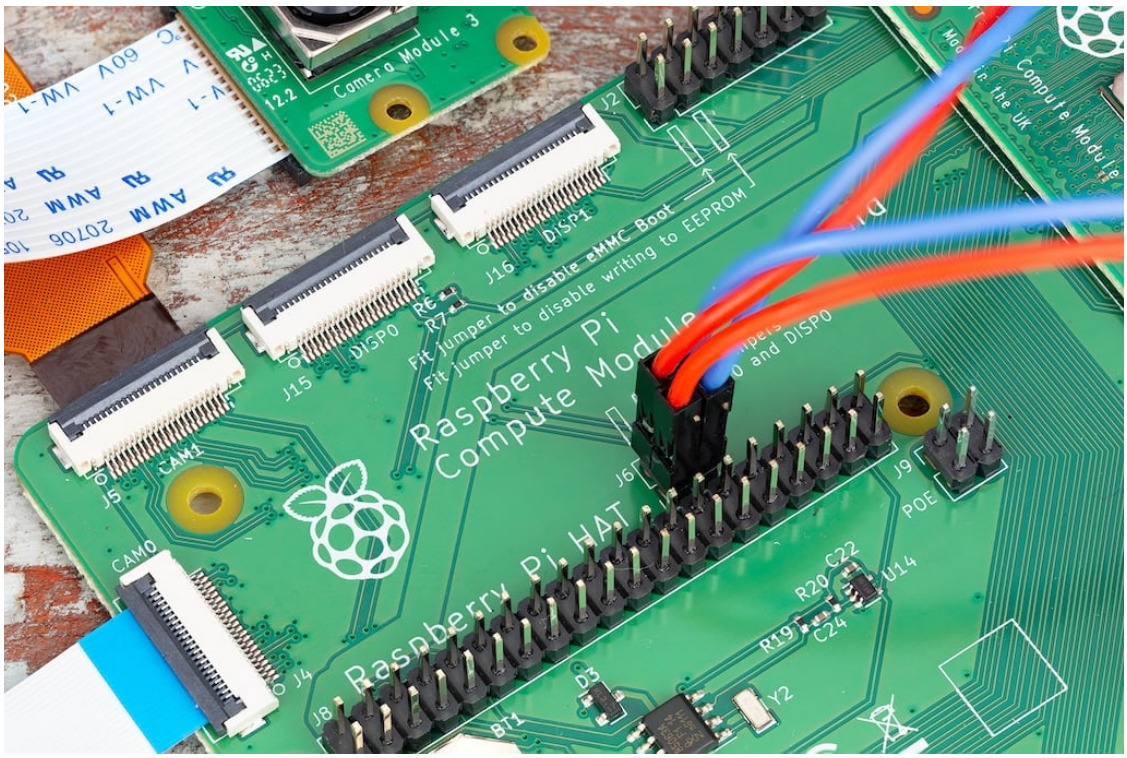
3. Connect the Camera Module to the CAM0 port using a RPI-CAMERA board or a Raspberry Pi Zero camera cable.



4. *(CM1, CM3, CM3+, and CM4S only)*: Connect the following GPIO pins with jumper cables:
    - `28` to `CD0_SDA`
    - `29` to `CD0_SCL`
    - `30` to `CAM0_I01`
    - `31` to `CAM0_I00`



5. *(CM4 only)*: Connect the J6 GPIO pins with two vertical-orientation jumpers.



6. Reconnect the Compute Module to power.

7. *(CM1, CM3, CM3+, and CM4S only)*: Add the following directive to `/boot/firmware/config.txt` to assign GPIO 31 as the CAM0 regulator:

```
dtparam=cam0_reg
```

8. Add the appropriate directive to `/boot/firmware/config.txt` to manually configure the driver for your camera model:

camera model	directive
v1 camera	`dtoverlay=ov5647,cam0`
v2 camera	`dtoverlay=imx219,cam0`
v3 camera	`dtoverlay=imx708,cam0`
HQ camera	`dtoverlay=imx477,cam0`
GS camera	`dtoverlay=imx296,cam0`

9. Reboot your Compute Module with `sudo reboot`.

10. Run the following command to check the list of detected cameras:

```
rpicam-hello --list
```

You should see both camera models, referred to by the driver directives in the table above, in the output.

Software

Raspberry Pi OS includes the `libcamera` library to help you take images with your Raspberry Pi.

Take a picture

Use the following command to immediately take a picture and save it to a file in PNG encoding using the `MMDDhhmmss` date format as a filename:

```
rpicam-still --datetime -e png
```

Use the `-t` option to add a delay in milliseconds. Use the `--width` and `--height` options to specify a width and height for the image.

Take a video

Use the following command to immediately start recording a 10 second long video and save it to a file with the h264 codec named `video.h264`:

```
rpicam-vid -t 10000 -o video.h264
```

Specify which camera to use

By default, `libcamera` always uses the camera with index `0` in the `--list-cameras` list. To specify a camera option, get an index value for each camera from the following command:

```
$ rpicam-hello --list-cameras
Available cameras
-----------------
0 : imx477 [4056x3040] (/base/soc/i2c0mux/i2c@1/imx477@1a)
    Modes: 'SRGGB10_CSI2P' : 1332x990 [120.05 fps - (696, 528)/2664x1980 crop]
           'SRGGB12_CSI2P' : 2028x1080 [50.03 fps - (0, 440)/4056x2160 crop]
                             2028x1520 [40.01 fps - (0, 0)/4056x3040 crop]
```

```
                            4056x3040 [10.00 fps - (0, 0)/4056x3040 crop]

  1 : imx708 [4608x2592] (/base/soc/i2c0mux/i2c@0/imx708@1a)
      Modes: 'SRGGB10_CSI2P' : 1536x864 [120.13 fps - (768, 432)/3072x1728 crop]
                               2304x1296 [56.03 fps - (0, 0)/4608x2592 crop]
                               4608x2592 [14.35 fps - (0, 0)/4608x2592 crop]
```

In the above output:

- `imx477` refers to a HQ camera with an index of `0`
- `imx708` refers to a v3 camera with an index of `1`

To use the HQ camera, pass its index (`0`) to the `--camera libcamera` option:

```
rpicam-hello --camera 0
```

To use the v3 camera, pass its index (`1`) to the `--camera libcamera` option:

```
rpicam-hello --camera 1
```

I2C mapping of GPIO pins

By default, the supplied camera drivers assume that CAM1 uses `i2c-10` and CAM0 uses `i2c-0`. Compute module I/O boards map the following GPIO pins to `i2c-10` and `i2c-0`:

I/O Board Model	**`i2c-10` pins**	**`i2c-0` pins**
CM4 I/O Board	GPIOs 44,45	GPIOs 0,1
CM1, CM3, CM3+, CM4S I/O Board	GPIOs 0,1	GPIOs 28,29

To connect a camera to the CM1, CM3, CM3+ and CM4S I/O Board, add the following directive to `/boot/firmware/config.txt` to accommodate the swapped pin assignment:

```
dtoverlay=cm-swap-i2c0
```

Alternative boards may use other pin assignments. Check the documentation for your board and use the following alternate overrides depending on your layout:

Swap	**Override**

Swap	Override
Use GPIOs 0,1 for i2c0	`i2c0-gpio0`
Use GPIOs 28,29 for i2c0 (default)	`i2c0-gpio28`
Use GPIOs 44&45 for i2c0	`i2c0-gpio44`
Use GPIOs 0&1 for i2c10 (default)	`i2c10-gpio0`
Use GPIOs 28&29 for i2c10	`i2c10-gpio28`
Use GPIOs 44&45 for i2c10	`i2c10-gpio44`

GPIO pins for shutdown

For camera shutdown, Device Tree uses the pins assigned by the `cam1_reg` and `cam0_reg` overlays.

The CM4 IO Board provides a single GPIO pin for both aliases, so both cameras share the same regulator.

The CM1, CM3, CM3+, and CM4S I/O Board provides no GPIO pin for `cam1_reg` and `cam0_reg`, so the regulators are disabled on those boards. However, you can enable them with the following directives in `/boot/firmware/config.txt`:

- `dtparam=cam1_reg`
- `dtparam=cam0_reg`

To assign `cam1_reg` and `cam0_reg` to a specific pin on a custom board, use the following directives in `/boot/firmware/config.txt`:

- `dtparam=cam1_reg_gpio=<pin number>`
- `dtparam=cam0_reg_gpio=<pin number>`

For example, to use pin 42 as the regulator for CAM1, add the directive `dtparam=cam1_reg_gpio=42` to `/boot/firmware/config.txt`.

These directives only work for GPIO pins connected directly to the SoC, not for expander GPIO pins.

Attaching the official 7-inch display

Edit this on GitHub

NOTE

These instructions are intended for advanced users. If anything is unclear, use the Raspberry Pi Compute Module forums for technical help.

Update your system software and firmware to the latest version before starting. Compute Modules mostly use the same process, but sometimes physical differences force changes for a particular model.

Connect a display to DISP1

NOTE

The Raspberry Pi Zero camera cable cannot be used as an alternative to the RPI-DISPLAY adapter. The two cables have distinct wiring.

To connect a display to DISP1:

1. Disconnect the Compute Module from power.
2. Connect the display to the DISP1 port on the Compute Module IO board through the 22W to 15W display adapter.
3. *(CM1, CM3, CM3+, and CM4S only)*: Connect the following GPIO pins with jumper cables:
    - `0` to `CD1_SDA`
    - `1` to `CD1_SCL`
4. Reconnect the Compute Module to power.
5. Add the following line to `/boot/firmware/config.txt`:

```
dtoverlay=vc4-kms-dsi-7inch
```

6. Reboot your Compute Module with `sudo reboot`. Your device should detect and begin displaying output to your display.

Connect a display to DISP0

To connect a display to DISP0:

1. Connect the display to the DISP0 port on the Compute Module IO board through the 22W to 15W display adapter.
2. *(CM1, CM3, CM3+, and CM4S only)*: Connect the following GPIO pins with jumper cables:
    - `28` to `CD0_SDA`
    - `29` to `CD0_SCL`
3. Reconnect the Compute Module to power.
4. Add the following line to `/boot/firmware/config.txt`:

```
dtoverlay=vc4-kms-dsi-7inch
```

5. Reboot your Compute Module with `sudo reboot`. Your device should detect and begin displaying output to your display.

Disable touchscreen

The touchscreen requires no additional configuration. Connect it to your Compute Module, and both the touchscreen element and display should work once successfully detected.

To disable the touchscreen element, but still use the display, add the following line to `/boot/firmware/config.txt`:

```
disable_touchscreen=1
```

Disable display

To entirely ignore the display when connected, add the following line to to `/boot/firmware/config.txt`:

```
ignore_lcd=1
```