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

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):

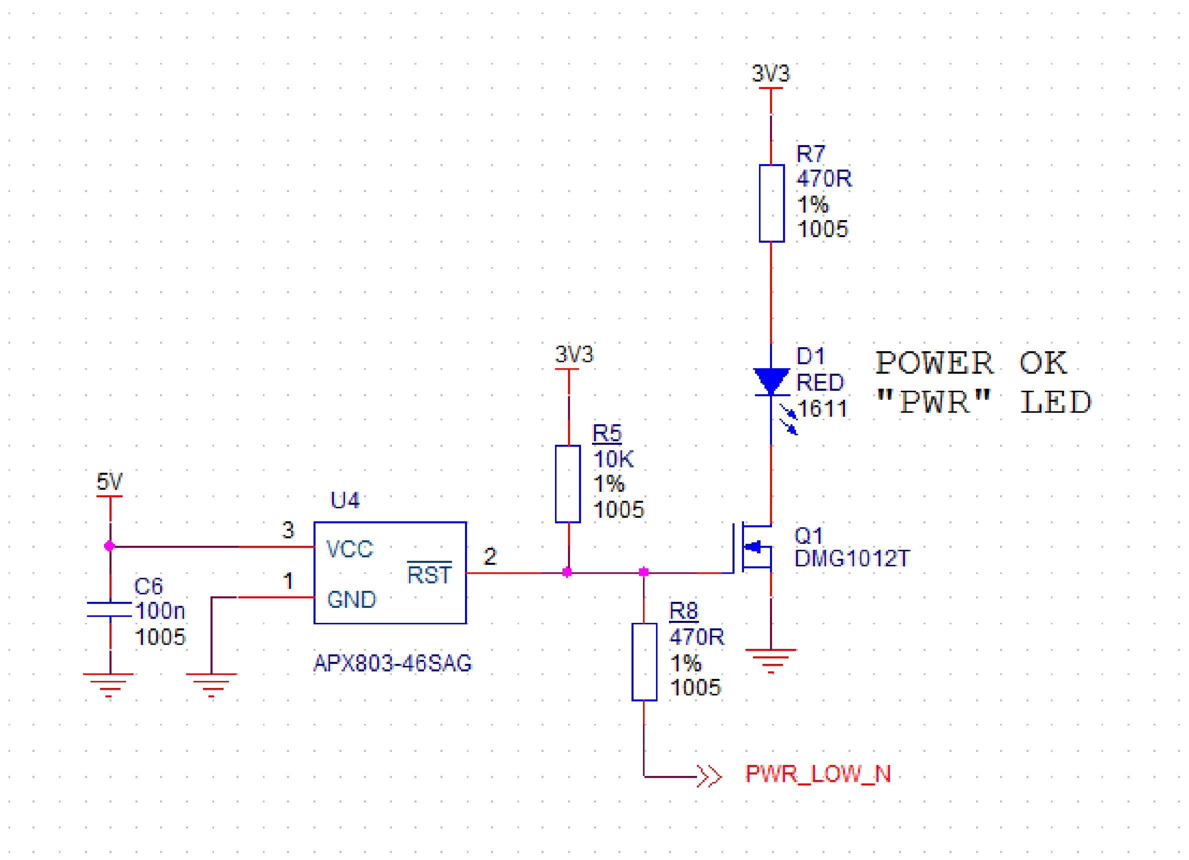
- CMIO Rev 3.0 (Supports CM1, CM3, CM3L, CM3+ and CM3+L)

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

- CMCDA Rev 1.1

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

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 32-bit and 64-bit, and Windows XP 32-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 remote/inaccessible products.

N.B. 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 NVMe / other storage devices.

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:-

- cd `usbboot/recovery`
- 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 `BOOT_ORDER=0xf2`
    - For SD/EMMC boot `BOOT_ORDER=0xf1`
    - For USB boot failing over to EMMC `BOOT_ORDER=0xf15`
- Run `./update-pieeprom.sh` to update the EEPROM image `pieeprom.bin` image file.

- If EEPROM write protection is required then 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 `raspi-gpio` package which is a tool for hacking and debugging GPIO

NOTE

You need to run `raspi-gpio` as root.

To install `raspi-gpio`:

```
sudo apt install raspi-gpio
```

If `apt` can't find the `raspi-gpio` package, you will need to do an update first:

```
sudo apt update
```

To get help on `raspi-gpio`, run it with the `help` argument:

```
sudo raspi-gpio help
```

For example, to see the current function and level of all GPIO pins use:

```
sudo raspi-gpio get
```

NOTE

`raspi-gpio` can be used with the `funcs` argument to get a list of all supported GPIO functions per pin. It will print out a table in CSV format. The idea is to pipe the table to a `.csv` file and then load this file using e.g. Excel:

```
sudo raspi-gpio funcs > gpio-funcs.csv
```

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 `raspi-gpio` to look at the setup of the GPIO pins to check that they are as you expect:

```
raspi-gpio get
```

If something seems to be going awry, useful information can also be found by dumping the GPU log messages:

```
sudo vcdbg log 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` 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/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/dt-blob.bin /boot/minimal-cm-dt-blob.dts
```

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

```
sudo dtc -@ -I dts -O dtb -o /boot/overlays/example1.dtbo /boot/example1-overla
y.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/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/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/config.txt` and add the 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:

```
sudo raspi-gpio get
```

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

Attaching a Raspberry Pi Camera Module

Edit this on GitHub

NOTE

These instructions are intended for advanced users, if anything is unclear please use the Raspberry Pi Camera forums for technical help.

Unless explicitly stated otherwise, these instructions will work identically on both the Compute Module 1 and Compute Module 3, attached to a Compute Module IO Board. Compute Module 4 is slightly different, so please refer to the appropriate section.

The Compute Module has two CSI-2 camera interfaces. CAM0 has two CSI-2 data lanes, whilst CAM1 has four data lanes. The Compute Module IO board exposes both of these interfaces. Note that the standard Raspberry Pi devices use CAM1, but only expose two data lanes.

Please note that the camera modules are **not** designed to be hot pluggable. They should always be connected or disconnected with the power off.

Updating your System

The camera software is under constant development. Please ensure your system is up to date prior to using these instructions.

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

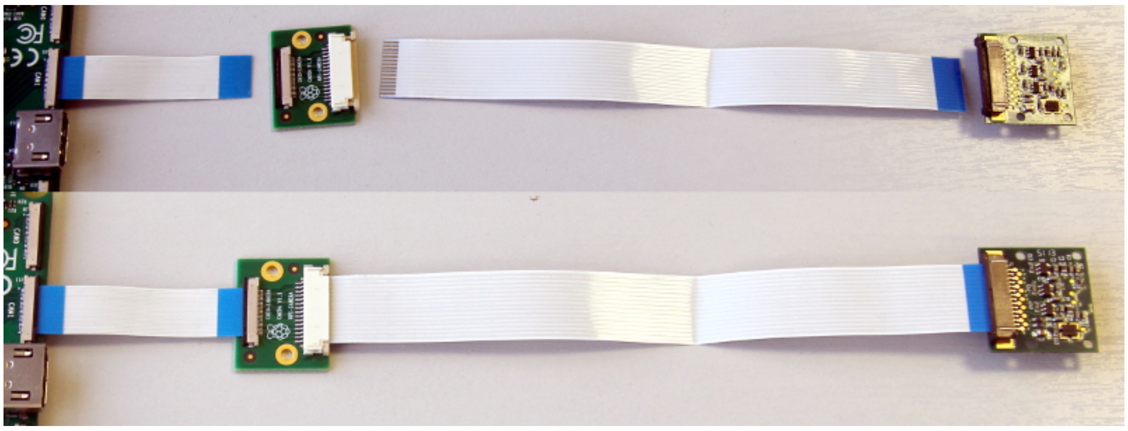
Crypto Chip

When using the Compute Module to drive cameras, it is NOT necessary to incorporate the crypto chip used on the Raspberry Pi—designed camera boards when attaching the OM5647, IMX219 or HQ Camera Modules directly to the Compute Module carrier board. The Raspberry Pi firmware will automatically detect the Compute Module and allow communications with the Camera Module to proceed without the crypto chip being present.

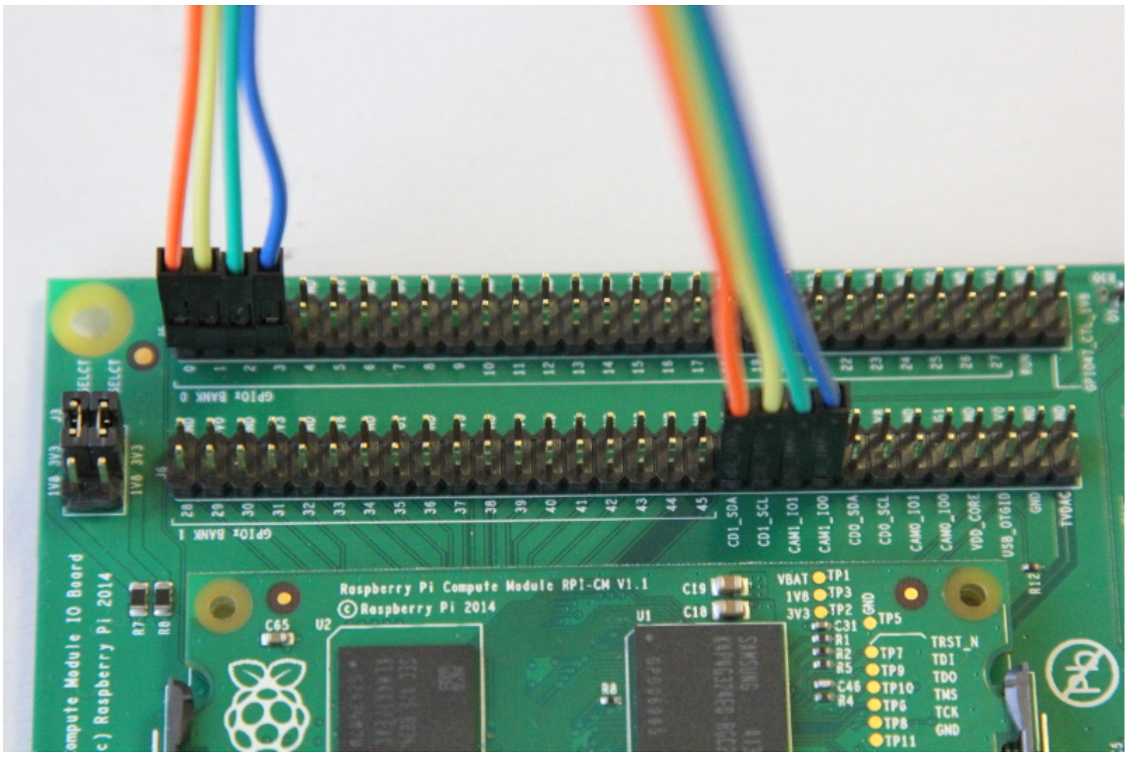
Quickstart Guide

To connect a single camera:

1. Power the Compute Module down.
2. On the Compute Module, run `sudo raspi-config` and enable the camera.
3. Connect the RPI-CAMERA board and Camera Module to the CAM1 port. As an alternative, the Raspberry Pi Zero camera cable can be used.



4. (CM1 & CM3 only) Connect GPIO pins together as shown below.



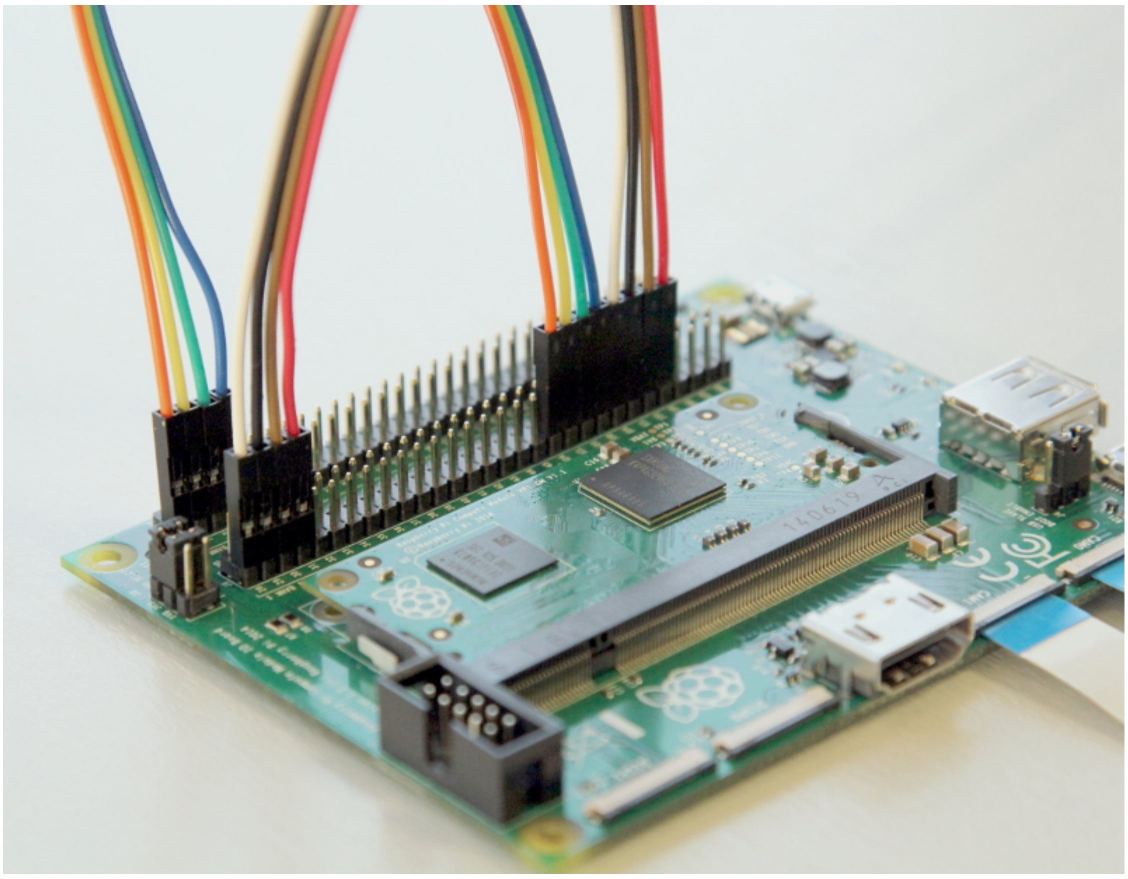
5. Power the Compute Module up and run `sudo wget https://datasheets.raspberrypi.com/cmio/dt-blob-cam1.bin -O /boot/dt-blob.bin`

6. Finally, reboot for the dt-blob.bin file to be read.

To connect two cameras, follow the steps as for a single camera and then also:

1. Whilst powered down, repeat step 3 with CAM0.

2. (CM1 and CM3 only) Connect the GPIO pins for the second camera.



3. (CM4 only) Add jumpers to J6.

4. Power up and run `sudo wget https://datasheets.raspberrypi.com/cmio/dt-blob-dualcam.bin -O /boot/dt-blob.bin`

5. Reboot for the dt-blob.bin file to be read.

NOTE

The default wiring uses GPIOs 2&3 to control the primary camera. These GPIOs can also be used for I2C, but doing so will result in a conflict, and the camera is unlikely to work. **Do not enable I2C via `dtparam=i2c_arm=on` if you wish to use the camera with the default wiring**

Software Support

The supplied camera applications `raspivid` and `raspistill` have the -cs (--camselect) option to specify which camera should be used.

If you are writing your own camera application based on the MMAL API you can use the MMAL_PARAMETER_CAMERA_NUM parameter to set the current camera. E.g.

```
MMAL_PARAMETER_INT32_T camera_num = {{MMAL_PARAMETER_CAMERA_NUM, sizeof(camera_
num)}, CAMERA_NUMBER};
status = mmal_port_parameter_set(camera->control, &camera_num.hdr);
```

Advanced Issues

The Compute Module IO board has a 22-way 0.5mm FFC for each camera port, with CAM0 being a two-lane interface and CAM1 being the full four-lane interface. The standard Raspberry Pi uses a 15-way 1mm FFC cable, so you will need either an adapter (part# RPI-CAMERA) or a Raspberry Pi Zero camera cable.

The CMIO board for Compute Modules 1 & 3 differ slightly in approach to that for Compute Module 4. They will be considered separately.

Compute Module 1 & 3

On the Compute Module IO board it is necessary to bridge the GPIOs and I2C interface required by the Raspberry Pi OS to the CAM1 connector. This is done by connecting the GPIOs from the J6 GPIO connector to the CD1_SDA/SCL and CAM1_IO0/1 pins on the J5 connector using jumper wires.

NOTE

The pin numbers below are provided only as an example. LED and SHUTDOWN pins can be shared by both cameras, if required.

The SDA and SCL pins must be either GPIOs 0 and 1, GPIOs 28 and 29, or GPIOs 44 and 45, and must be individual to each camera.

Steps to attach a Raspberry Pi Camera (to CAM1)

1. Attach the 0.5mm 22W FFC flexi (included with the RPI-CAMERA board) to the CAM1 connector (flex contacts face down). As an alternative, the Raspberry Pi Zero camera cable can be used.
2. Attach the RPI-CAMERA adaptor board to the other end of the 0.5mm flex (flex contacts face down).
3. Attach a Raspberry Pi Camera to the other, larger 15W 1mm FFC on the RPI-CAMERA adaptor board (**contacts on the Raspberry Pi Camera flex must face up**).
4. Attach CD1_SDA (J6 pin 37) to GPIO0 (J5 pin 1).
5. Attach CD1_SCL (J6 pin 39) to GPIO1 (J5 pin 3).
6. Attach CAM1_IO1 (J6 pin 41) to GPIO2 (J5 pin 5).
7. Attach CAM1_IO0 (J6 pin 43) to GPIO3 (J5 pin 7).

Note, the numbers in brackets are conventional, physical pin numbers, numbered from left to right, top to bottom. The numbers on the silkscreen correspond to the Broadcom SoC GPIO numbers.

Steps to attach a second Raspberry Pi Camera (to CAM0)

Attach the second camera to the (CAM0) connector as before.

Connect up the I2C and GPIO lines.

1. Attach CD0_SDA (J6 pin 45) to GPIO28 (J6 pin 1).
2. Attach CD0_SCL (J6 pin 47) to GPIO29 (J6 pin 3).
3. Attach CAM0_IO1 (J6 pin 49) to GPIO30 (J6 pin 5).
4. Attach CAM0_IO0 (J6 pin 51) to GPIO31 (J6 pin 7).

Compute Module 4

On the Compute Module 4 IO board the CAM1 connector is already wired to the I2C on GPIOs 44 & 45, and the shutdown line is connected to GPIO 5 on the GPIO expander. There is no LED signal wired through. No hardware changes are required to use CAM1 other than connecting the 22pin FFC to the CAM1 connector (flex contacts face down).

To connect a second Raspberry Pi camera (to CAM0), two jumpers must be added to J6 in a vertical orientation. The CAM0 connector shares the shutdown line with CAM1.

Configuring default pin states (all CM variants)

The GPIOs that we are using for the camera default to input mode on the Compute Module. To override these default settings and also tell the system that these are the pins to be used by the camera, we need to create a `dt-blob.bin` that is loaded by the firmware when the system boots up. This file is built from a source dts file that contains the required settings, and placed on the boot partition.

Sample device tree source files are provided at the bottom of this document. These use the default wiring as described in this page.

The `pin_config` section in the `pins_cm { }` (Compute Module 1), `pins_cm3 { }` (Compute Module 3), or `pins_cm4 { }` (Compute Module 4) section of the source dts needs the camera's LED and power enable pins set to outputs:

```
pin@p2  { function = "output"; termination = "no_pulling"; };
pin@p3  { function = "output"; termination = "no_pulling"; };
```

To tell the firmware which pins to use and how many cameras to look for, add the following to the `pin_defines` section:

```
pin_define@CAMERA_0_LED { type = "internal"; number = <2>; };
pin_define@CAMERA_0_SHUTDOWN { type = "internal"; number = <3>; };
pin_define@CAMERA_0_UNICAM_PORT { type = "internal"; number = <1>; };
pin_define@CAMERA_0_I2C_PORT { type = "internal"; number = <0>; };
pin_define@CAMERA_0_SDA_PIN { type = "internal"; number = <0>; };
pin_define@CAMERA_0_SCL_PIN { type = "internal"; number = <1>; };
```

Indentation and line breaks are not critical, so the example files expand these blocks out for readability.

The Compute Module's **pin_config** section needs the second camera's LED and power enable pins configured:

```
pin@p30 { function = "output"; termination = "no_pulling"; };
pin@p31 { function = "output"; termination = "no_pulling"; };
```

In the Compute Module's **pin_defines** section of the dts file, change the **NUM_CAMERAS** parameter to 2 and add the following:

```
pin_define@CAMERA_1_LED { type = "internal"; number = <30>; };
pin_define@CAMERA_1_SHUTDOWN { type = "internal"; number = <31>; };
pin_define@CAMERA_1_UNICAM_PORT { type = "internal"; number = <0>; };
pin_define@CAMERA_1_I2C_PORT { type = "internal"; number = <0>; };
pin_define@CAMERA_1_SDA_PIN { type = "internal"; number = <28>; };
pin_define@CAMERA_1_SCL_PIN { type = "internal"; number = <29>; };
```

Sample device tree source files

Enable CAM1 only

Enable CAM1 and CAM0

Compiling a DTS file to a device tree blob

Once all the required changes have been made to the `dts` file, it needs to be compiled and placed on the boot partition of the device.

Instructions for doing this can be found on the Pin Configuration page.

Attaching the Official 7-inch Display

Edit this on GitHub

NOTE

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

Please ensure your system software is updated before starting. Largely speaking the approach taken for Compute Modules 1, 3, and 4 is the same, but there are minor differences in physical setup required. It will be indicated where a step applies only to a specific platform.

WARNING

The Raspberry Pi Zero camera cable cannot be used as an alternative to the RPI-DISPLAY adaptor, because its wiring is different.

WARNING

Please note that the display is **not** designed to be hot pluggable. It (and camera modules) should always be connected or disconnected with the power off.

Quickstart Guide (Display Only)

Connecting to DISP1

1. Connect the display to the DISP1 port on the Compute Module IO board through the 22W to 15W display adaptor.
2. (CM1 and CM3 only) Connect these pins together with jumper wires:

```
GPIO0 - CD1_SDA
GPIO1 - CD1_SCL
```

3. Power up the Compute Module and run:

```
sudo wget https://datasheets.raspberrypi.com/cmio/dt-blob-disp1-only.bin -O /boot/dt-blob.bin
```

4. Reboot for the `dt-blob.bin` file to be read.

Connecting to DISP0

1. Connect the display to the DISP0 port on the Compute Module IO board through the 22W to 15W display adaptor.
2. (CM1 and CM3 only) Connect these pins together with jumper wires:

```
GPIO28 - CD0_SDA
GPIO29 - CD0_SCL
```

3. Power up the Compute Module and run:

   ```
   sudo wget https://datasheets.raspberrypi.com/cmio/dt-blob-disp0-only.bin -O
   /boot/dt-blob.bin
   ```

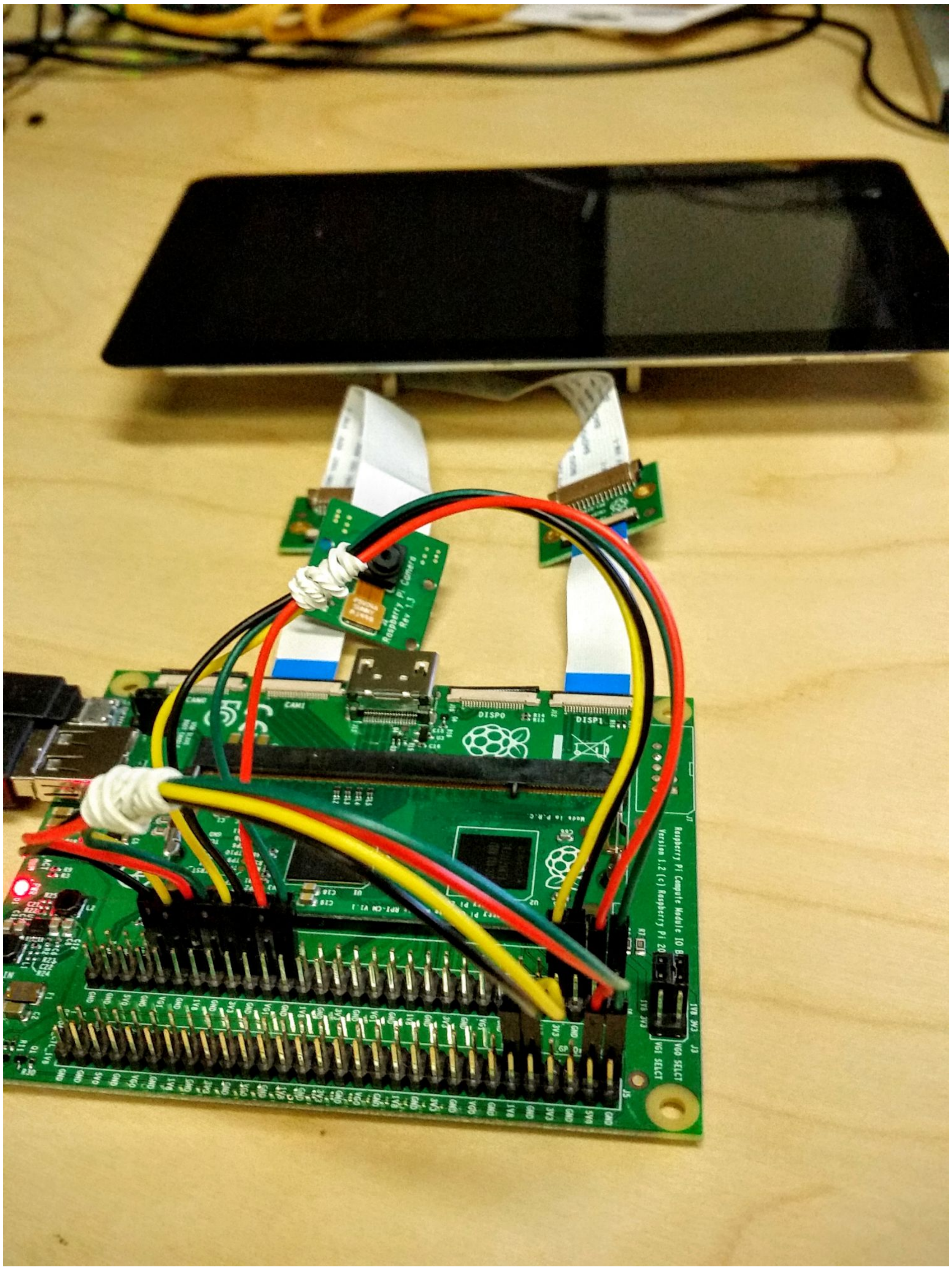
4. Reboot for the `dt-blob.bin` file to be read.

Quickstart Guide (Display and Cameras)

To enable the display and one camera:*

1. Connect the display to the DISP1 port on the Compute Module IO board through the 22W to 15W display adaptor, called RPI-DISPLAY.
2. Connect the Camera Module to the CAM1 port on the Compute Module IO board through the 22W to 15W adaptor called RPI-CAMERA. Alternatively, the Raspberry Pi Zero camera cable can be used.
3. (CM1 and CM3 only) Connect these pins together with jumper wires:

   ```
   GPIO0 - CD1_SDA
   GPIO1 - CD1_SCL
   GPIO2 - CAM1_IO1
   GPIO3 - CAM1_IO0
   ```



(Please note this image needs to be updated to have the extra jumper leads removed and use the standard wiring (2&3 not 4&5))

4. Power up the Compute Module and run:

```
sudo wget https://datasheets.raspberrypi.com/cmio/dt-blob-disp1-cam1.bin -O
/boot/dt-blob.bin
```

5. Reboot for the `dt-blob.bin` file to be read.

To enable the display and both cameras:*

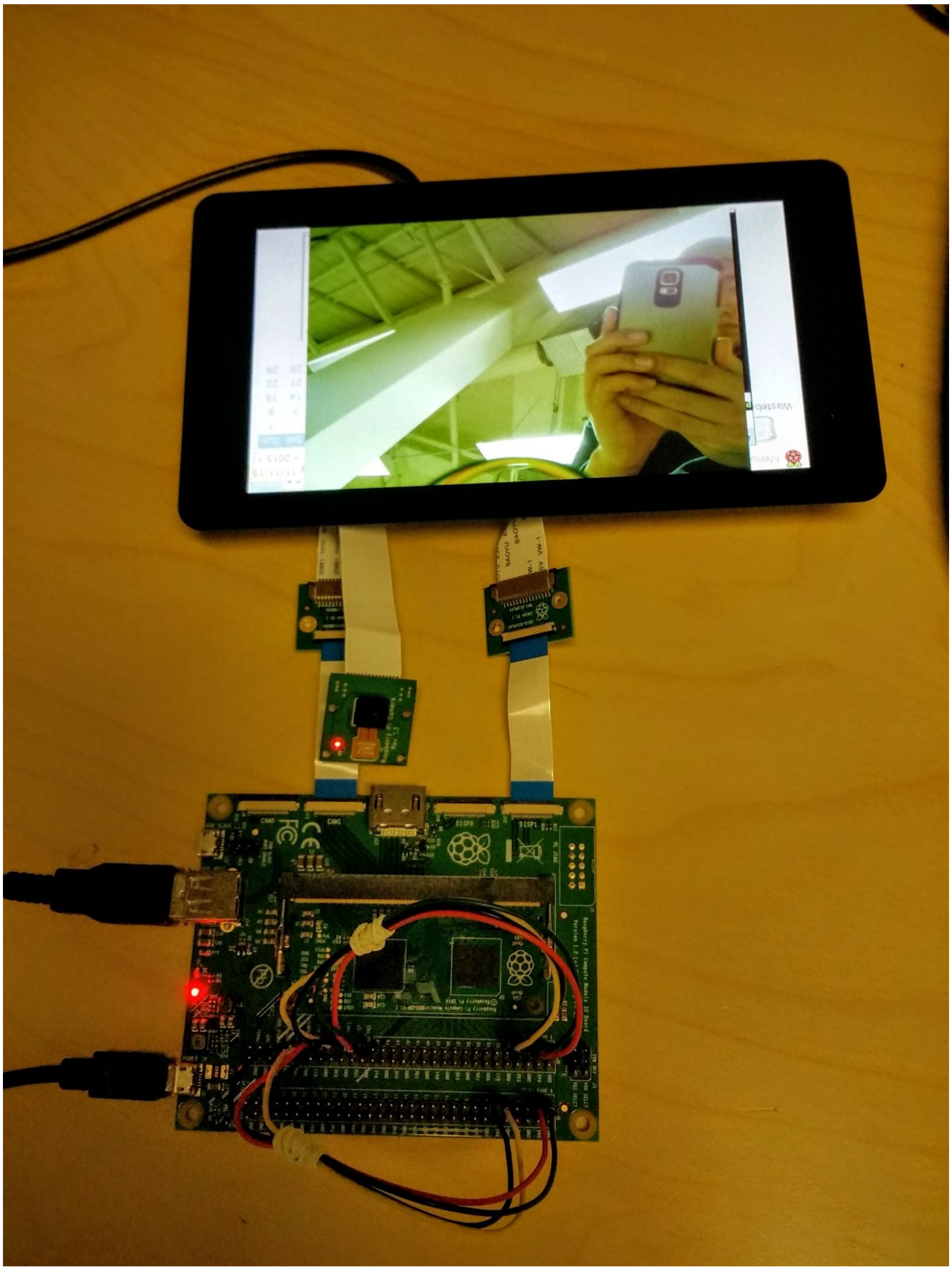
1. Follow the steps for connecting the display and one camera above.
2. Connect the Camera Module to the CAM0 port on the Compute Module IO board through the 22W to 15W adaptor called RPI-CAMERA. Alternatively, the Raspberry Pi Zero camera cable can be used.
3. (CM1 and CM3 only) Add links:

   ```
   GPIO28 - CD0_SDA
   GPIO29 - CD0_SCL
   GPIO30 - CAM0_IO1
   GPIO31 - CAM0_IO0
   ```

4. (CM4 only) Add jumpers to J6.
5. Power up the Compute Module and run:

   ```
   sudo wget https://datasheets.raspberrypi.com/cmio/dt-blob-disp1-cam2.bin -O
   /boot/dt-blob.bin
   ```

6. Reboot for the `dt-blob.bin` file to be read.



(Please note this image needs to be updated to show two Camera Modules and the standard wiring)

Software Support

There is no additional configuration required to enable the touchscreen. The touch interface should work out of the box once the screen is successfully detected.

If you wish to disable the touchscreen element and only use the display side, you can add the command `disable_touchscreen=1` to /boot/config.txt to do so.

To make the firmware to ignore the display even if connected, then add `ignore_lcd=1` to /boot/config.txt.

Firmware Configuration

The firmware looks at the dt-blob.bin file for the relevant configuration to use for the screen. It looks at the pin_number@ defines for

```
DISPLAY_I2C_PORT
DISPLAY_SDA
DISPLAY_SCL
DISPLAY_DSI_PORT
```

The I2C port, SDA and SCL pin numbers are self explanatory. DISPLAY_DSI_PORT selects between DSI1 (the default) and DSI0.

Once all the required changes have been made to the `dts` file, it needs to be compiled and placed on the boot partition of the device.

Instructions for doing this can be found on the Pin Configuration page.

Sources

- dt-blob-disp1-only.dts
- dt-blob-disp1-cam1.dts
- dt-blob-disp1-cam2.dts
- dt-blob-disp0-only.dts (Uses wiring as for CAM0)