Using I²C on Raspberry Pi SBCs

Colophon

Release	1
Build date	27/05/2026
Build version	9184d1f8ef18

Legal disclaimer notice

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME ("RESOURCES") ARE PROVIDED BY RASPBERRY PI LTD ("RPL") "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RPL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.

The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPL or other third party intellectual property right.

HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). RPL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

Raspberry Pi products are provided subject to RPL's Standard Terms. RPL's provision of the RESOURCES does not expand or otherwise modify RPL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Document version history

Release	Date	Description
1	27 May 2026	Initial release

Scope of document

This document applies to the following Raspberry Pi products:

Single Board Computers / SBCs

Pi Zero			Pi Zero 2		Pi 1				Pi 2	Pi 3			Pi 4	Pi 5
-	W	H	W	WH	A	B	A+	B+	B	A+	B	B+	B	-
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Compute Modules

CM0	CM1	CM3	CM3+	CM4	CM4S	CM5
✓	✓	✓	✓	✓	✓	✓

Keyboard Computers

Pi 400	Pi 500	Pi 500+
✓	✓	✓

Introduction

All Raspberry Pi single-board computers, compute modules, and keyboard computers (Raspberry Pi 400, Raspberry Pi 500, Raspberry Pi 500+) support a protocol called I²C. This is a serial protocol that is commonly used to attach peripheral devices.

This white paper will explain how the I²C peripherals on Raspberry Pi devices are arranged and how to use them. It will also cover bit-banged I²C.

All of the information in this white paper is based on the Trixie version of Raspberry Pi OS, which is the latest version at the time of writing.

What is I²C?

I²C is a serial communication protocol used to connect multiple electronic components — such as microcontrollers, sensors, and memory chips — using only two wires plus a common ground:

- **SDA (serial data line)** Carries data
- **SCL (serial clock line)** Carries the clock signal

It was developed by Philips (now NXP) to enable simple, short-distance communication between devices on the same circuit board. The Broadcom implementation of I²C on Raspberry Pi Zero to Raspberry Pi 4 Model B is called the Broadcom Serial Control (BSC).

Key features

Controller/target architecture (formerly master-slave)
: One or more controllers control communication; targets respond when addressed

Address-based communication
: Each target device has a unique 7-bit or 10-bit address

Shared bus
: Multiple devices can use the same two wires

Synchronous protocol
: Data transfer is synchronised by the clock line

Pull-up resistors required
: Both SDA and SCL are open-drain lines

Typical speeds

Standard mode
: 100kb/s

Fast mode
: 400kb/s

Fast mode plus
: 1Mb/s

High-speed mode
: 3.4Mb/s

Common uses

- Sensors (temperature, pressure, IMUs)
- EEPROMs and RTCs
- Configuration and control interfaces
- Communication between microcontrollers and peripherals

In short, I²C is a simple, low-pin-count, low-speed communication protocol ideal for short-distance device communication in embedded systems.

SMBus versus I²C

The System Management Bus (SMBus) is a subset of I²C. It shares the same two-wire (SDA/SCL) foundation, but with stricter rules for timing, electricals, and protocols. I²C is faster and more flexible (up to 3.4MHz), while SMBus prioritises robust, low-speed management (10–100kHz), with features like timeouts and defined voltage levels for system monitoring (such as battery management). The key differences of SMBus include its mandatory 10kHz-minimum clock (versus 0Hz for I²C), a 35ms timeout, and fixed voltage levels; I²C, on the other hand, can operate slower or faster, and uses VDD-relative levels. This often makes them compatible, but not always interchangeable.

Further reading

The I²C specification: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

The SMBus specification: https://www.smbus.org/specs/SMBus_3_0_20141220.pdf

I²C on Raspberry Pi devices

Implementation of I²C and the available features depends on which model of Raspberry Pi you are using. Later boards provide more facilities.

Warning

Raspberry Pi devices use 3.3V signalling levels. If your peripherals use 5V signalling on the GPIO I²C pins, you will need to use a level shifter to convert the Raspberry Pi's signals to 5V.

Table 1.
Available I²C ports

Model	Number of available ports	Location of peripheral
Raspberry Pi Zero, 1, 2, 3	2 (+1 HDMI)	BCM2835, BCM2836, BCM2837
Raspberry Pi 4 Model B	6 (+2 HDMI)	BCM2711
Raspberry Pi Compute Module 4	6 (+2 HDMI)	BCM2711
Raspberry Pi 5	4 (+2 HDMI)	BCM2712 + RP1
Raspberry Pi Compute Module 5	5 (+2 HDMI)	BCM2712 + RP1

Note

Readers of the datasheets (see Table 2) provided for the system on chip (SoC) used in each device will notice that there appears to be a mismatch between the number of ports supported by the SoC and the number of ports available on the devices that use that particular SoC. This is down to a lack of GPIO pin locations — not all I²C ports can be exported to the Raspberry Pi form factor.

Table 1 shows that the I²C peripherals on Raspberry Pi 5 and Raspberry Pi Compute Module 5 are split between the SoC and the RP1 coprocessor. The Linux I²C driver insulates the end user from this difference. A later section will go into more detail on the split.

Bit-banged I²C

Although using the dedicated hardware peripherals for I²C is the best option, as I²C at its simplest is just two lines moving between 0 and 3.3V, there is also an option to drive the GPIO pins using a completely software-based approach — this is known as 'bit banging'.

There are limitations to this, however, the major one being the speed of the response to interrupts and incoming data, along with keeping the outgoing waveforms within the timing specifications. While hardware implementations include features like FIFO and queues to handle the required timing — making high speeds possible — software implementations are limited by the speed of the CPU and the latencies introduced by the operating system.

That said, since I²C is a relatively slow communications system, bit-banged drivers are quite feasible; these will be covered later.

I²C and GPIOs

I²C peripherals need to be assigned to the GPIO pins so that the signals can be used off the board. However, the range of GPIOs that each I²C peripheral can be attached to is limited.

The datasheets for each SoC have tables of possible GPIO assignments, covering all peripherals as well as I²C. These can be found in the Raspberry Pi Product Information Portal and are linked in the following table:

Table 2.
Datasheets

Model	Datasheet link	GPIO function select section
Raspberry Pi Zero, 1, 2, 3	https://pip-assets.raspberrypi.com/categories/579-raspberry-pi-zero/documents/RP-008249-DS-1-bcm2835-peripherals.pdf	6.2 Alternative Function Assignments
Raspberry Pi 4 Model B, Raspberry Pi Compute Module 4	https://pip-assets.raspberrypi.com/categories/545-raspberry-pi-4-model-b/documents/RP-008248-DS-1-bcm2711-peripherals.pdf	5.3. Alternative Function Assignments
Raspberry Pi 5, Raspberry Pi Compute Module 5	https://pip-assets.raspberrypi.com/categories/892-raspberry-pi-5/documents/RP-008370-DS-1-rp1-peripherals.pdf	3.1.1. Function select

When the Linux kernel starts, device drivers are attached to each I²C peripheral as follows:

Table 3.
Device assignment of ports

Model	Broadcom/RP1 port name	Default Linux device names (/dev/*)
Raspberry Pi Zero, 1, 2, 3	BSC0-2; BSC2 reserved for HDMI	i2c-0, i2c-1
Raspberry Pi 4 Model B, Raspberry Pi Compute Module 4	BSC0-6; BSC2 not used on BCM2711 devices	i2c-1, i2c-20 (HDMI), i2c-21 (HDMI)
Raspberry Pi 5, Raspberry Pi Compute Module 5	I2C0-6	i2c-1, i2c-13 (HDMI), i2c-14 (HDMI)

Tip

`pinctrl funcs`, installed as part of Raspberry Pi OS, is a command line way to discover which functions are available on which GPIOs for the current platform.

Cameras and displays

On Raspberry Pi devices, there are up to two MIPI CSI/DSI ports used for communication with cameras and LCD displays respectively. These ports have an I²C peripheral allocated to them. These are standard buses and can be used in the normal fashion.

On most Raspberry Pi CMs, these ports can be used but as they do not connect to the standard GPIO header on Raspberry Pi devices, it is not possible to access these signals unless you are using a Raspberry Pi CM on which they are available on the board connector.

On the Raspberry Pi Compute Module 4 IO Board the CAM0 and DSI0 I²C ports use GPIO 0 & GPIO1 which are available on the standard header.

HDMI

On the BCM2711 and BCM2712 there are dedicated I²C peripherals assigned to the HDMI ports. These are used to transfer EDID (Extended Display Identification Data) information from the HDMI device to the Raspberry Pi SBC.

If the HDMI ports are not being used then these I²C peripherals can also be made available for general use. Whilst these signals are available on the HDMI connectors of Raspberry Pi SBCs, they are easier to access when using Raspberry Pi CMs.

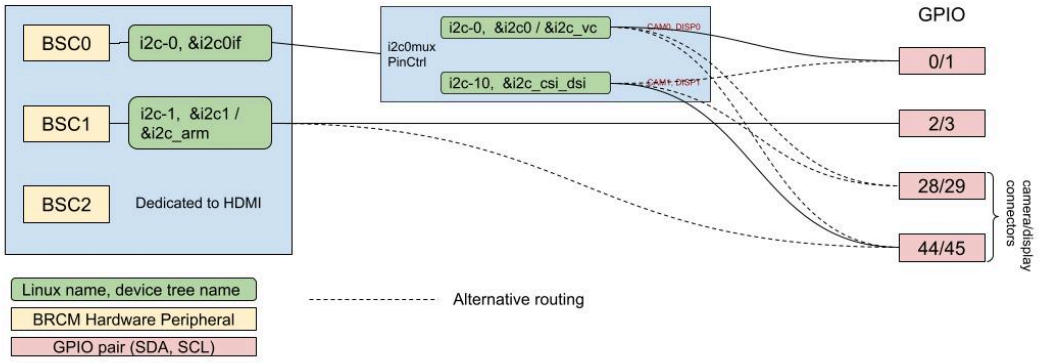
Note

HDMI I²C ports are 5v tolerant as required by the HDMI specification

Connection diagrams for I²C

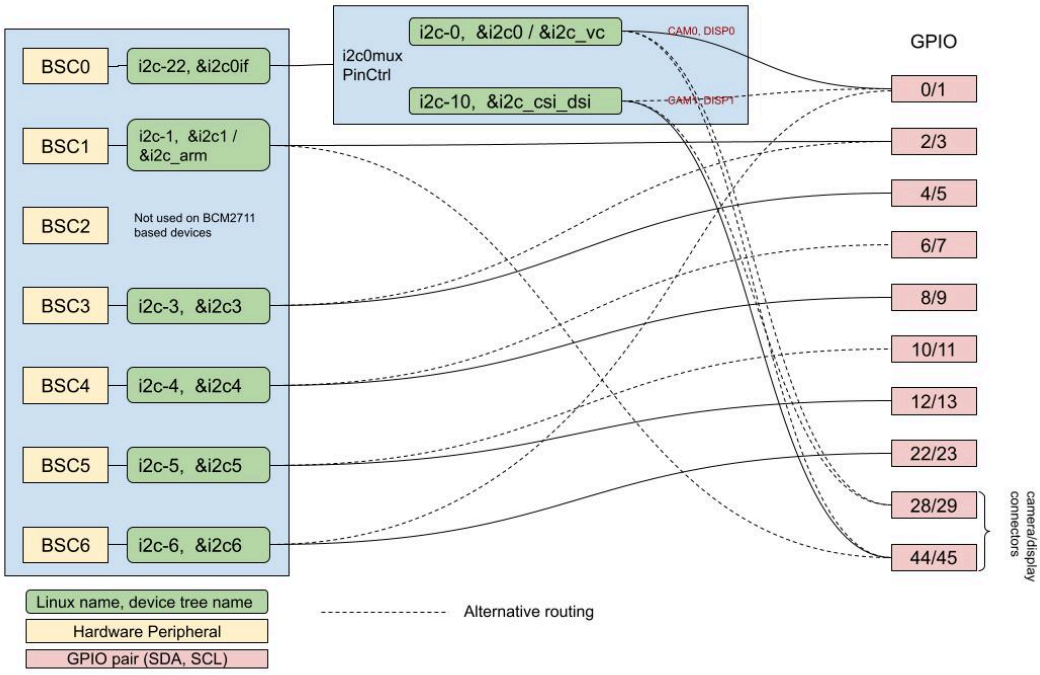
Raspberry Pi Zero to Raspberry Pi 3 Model B+

Figure 1.
Device ports on Raspberry Pi Zero to Raspberry Pi 3 Model B+



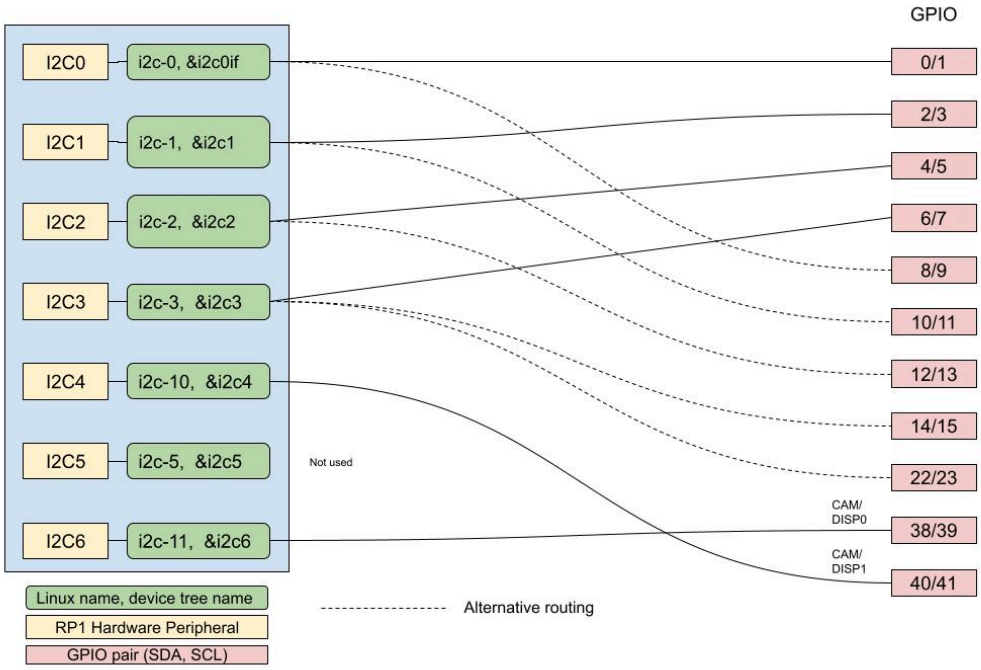
Raspberry Pi 4 Model B and Raspberry Pi Compute Module 4

Figure 2.
Device ports on Raspberry Pi 4 Model B and Raspberry Pi Compute Module 4



Raspberry Pi 5 and Raspberry Pi Compute Module 5

Figure 3.
Device ports on Raspberry Pi 5 and Raspberry Pi Compute Module 5



On-board Pull-ups

On all devices, GPIO 2 and GPIO 3, which are those used by default for I2C1, have 1.8k pull-ups soldered to the board. For all other GPIOs used for I²C, additional pull-ups may be required.

Setting up a Raspberry Pi for I²C

Enabling

The easiest way to enable I²C–1 (GPIO 2&3) on Raspberry Pi devices is to use the Control Centre application provided in Raspberry Pi OS.

Select 'Preferences', 'Control Centre' from the main desktop menu. Next, select 'Interfaces' from the left-hand column and enable the I²C option. Behind the scenes, the Control Centre application will add/update the `config.txt` file (`/boot/firmware/config.txt`) with the following entry:

```
dtparam=i2c_arm=on
```

Now if you look at the devices (`ls /dev/i2c*`), you will see that a new device has appeared: `/dev/i2c-1`. Note that there are also other I²C devices present; we will come to those later.

Device tree options

I²C devices are allocated to specific GPIOs (see Table 4) by default, but it is possible to direct specific I²C peripherals to specific GPIOs using device tree overlays. These commands are usually inserted into the `config.txt` file, where they are passed on to the kernel, which in turn sets up the required pin muxing.

Warning

You cannot arbitrarily assign I²C ports to any GPIOs.

Warning

For historical reasons, many overlays use the terms `pin` and `GPIO` interchangeably, but they always refer to GPIO numbers, not to pins on the 40-pin header.

The following table lists the device tree overlays that are applicable to I²C. The description column gives the results of running `dtoverlay -h <name of overlay>`:

Table 4.

Device tree overlays

<table>
<tr><th>Overlay</th><th>Description</th><th>Parameters</th></tr>
<tr><td>i2c0</td><td>Change i2c0 pin usage. Not all pin combinations are usable on all platforms — excluding our Compute Module range, Raspberry Pi platforms can only use this to disable transaction combining. Do NOT use it in conjunction with `dtparam=i2c_vc=on`. The base DT includes `i2c_mux_pinctrl`, which exposes two muxings of BSC0: GPIOs 0 and 1, and whichever combination is used for the camera and display connectors. This overlay disables that mux and configures `/dev/i2c0` to point to whichever set of pins is requested. `dtparam=i2c_vc=on` will try to enable the mux, so combining the two will cause conflicts.</td><td><table><tr><th>Parameter</th><th>Description</th></tr><tr><td>pins_0_1</td><td>Use pins 0 and 1 (default)</td></tr><tr><td>pins_28_29</td><td>Use pins 28 and 29</td></tr><tr><td>pins_44_45</td><td>Use pins 44 and 45</td></tr><tr><td>pins_46_47</td><td>Use pins 46 and 47</td></tr><tr><td>combine</td><td>Allow transactions to be combined (default: yes)</td></tr></table></td></tr>
<tr><td>i2c0-pi5[1]</td><td>Enable i2c0 (Raspberry Pi 5 only)</td><td><table><tr><th>Parameter</th><th>Description</th></tr><tr><td>pins_0_1</td><td>Use pins 0 and 1 (default)</td></tr><tr><td>pins_8_9</td><td>Use pins 8 and 9</td></tr><tr><td>baudrate</td><td>Set the baudrate for the interface (default: 100000)</td></tr></table></td></tr>
<tr><td>i2c1</td><td>Change i2c1 pin usage. Not all pin combinations are usable on all platforms — excluding our Compute Module range, Raspberry Pi platforms can only use this to disable transaction combining.</td><td><table><tr><th>Parameter</th><th>Description</th></tr><tr><td>pins_2_3</td><td>Use pins 2 and 3 (default)</td></tr><tr><td>pins_44_45</td><td>Use pins 44 and 45</td></tr><tr><td>combine</td><td>Allow transactions to be combined (default: yes)</td></tr></table></td></tr>
<tr><td>i2c1-pi5[1]</td><td>Enable i2c1 (Raspberry Pi 5 only)</td><td><table><tr><th>Parameter</th><th>Description</th></tr><tr><td>pins_2_3</td><td>Use pins 2 and 3 (default)</td></tr><tr><td>pins_10_11</td><td>Use pins 10 and 11</td></tr><tr><td>baudrate</td><td>Set the baudrate for the interface (default: 100000)</td></tr></table></td></tr>
<tr><td>i2c2-pi5[1]</td><td>Enable i2c2 (Raspberry Pi 5 only)</td><td><table><tr><th>Parameter</th><th>Description</th></tr><tr><td>pins_4_5</td><td>Use pins 4 and 5 (default)</td></tr><tr><td>pins_12_13</td><td>Use pins 12 and 13</td></tr><tr><td>baudrate</td><td>Set the baudrate for the interface (default: 100000)</td></tr></table></td></tr>
</table>

<table>
<tr><th>Overlay</th><th>Description</th><th>Parameters</th></tr>
<tr><td>i2c3</td><td>Enable the i2c3 bus (BCM2711 only)</td><td>
<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>pins_2_3</td><td>Use pins 2 and 3 (default)</td></tr>
<tr><td>pins_4_5</td><td>Use pins 4 and 5</td></tr>
<tr><td>baudrate</td><td>Set the baudrate for the interface (default: 100000)</td></tr>
</table>
</td></tr>
<tr><td>i2c3-pi5[1]</td><td>Enable i2c3 (Raspberry Pi 5 only)</td><td>
<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>pins_6_7</td><td>Use pins 6 and 7 (default)</td></tr>
<tr><td>pins_14_15</td><td>Use pins 14 and 15</td></tr>
<tr><td>pins_22_23</td><td>Use pins 22 and 23</td></tr>
<tr><td>baudrate</td><td>Set the baudrate for the interface (default: 100000)</td></tr>
</table>
</td></tr>
<tr><td>i2c4</td><td>Enable the i2c4 bus (BCM2711 only)</td><td>
<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>pins_6_7</td><td>Use pins 6 and 7</td></tr>
<tr><td>pins_8_9</td><td>Use pins 8 and 9 (default)</td></tr>
<tr><td>baudrate</td><td>Set the baudrate for the interface (default: 100000)</td></tr>
</table>
</td></tr>
<tr><td>i2c5</td><td>Enable the i2c5 bus (BCM2711 only)</td><td>
<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>pins_10_11</td><td>Use pins 10 and 11</td></tr>
<tr><td>pins_12_13</td><td>Use pins 12 and 13 (default)</td></tr>
<tr><td>baudrate</td><td>Set the baudrate for the interface (default: 100000)</td></tr>
</table>
</td></tr>
<tr><td>i2c6</td><td>Enable the i2c6 bus (BCM2711 only)</td><td>
<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>pins_0_1</td><td>Use pins 0 and 1</td></tr>
<tr><td>pins_22_23</td><td>Use pins 22 and 23 (default)</td></tr>
<tr><td>baudrate</td><td>Set the baudrate for the interface (default: 100000)</td></tr>
</table>
</td></tr>
</table>

[1] The selection of `-pi5` overlays is automatic, based on `overlay_map.dts`. It is recommended that users do NOT include the suffix in the `dtoverlay` command.

Bit-banging I²C to any GPIO

Device tree overlays can also be used to assign any spare GPIOs you like to the bit-banged driver.

```
dtoverlay=i2c-gpio,i2c_gpio_sda=<gpio>,i2c_gpio_scl=<gpio>,i2c_gpio_delay_us=<delay>,bus=<bus>
```

This will create a standard `/dev/i2cX` device.

Parameters

i2c_gpio_sda GPIO used for I²C data (default: 23)
i2c_gpio_scl GPIO used for I²C clock (default: 24)
i2c_gpio_delay_us Clock delay in microseconds (default: 2, 100kHz)
bus Set to a unique, non-zero value to get multiple `i2c-gpio` buses. If set, this number will be used as the preferred bus number (/dev/i2c-<n>). If not set, the default value is 0, but the bus number will be dynamically assigned — likely 3.

You can have as many of these devices as required, up to the total number of GPIOs available, although performance may be CPU limited.

Note

The internal 50k pull-ups on the GPIOs will be enabled automatically on the GPIO lines used for SDA and SCL (as I²C lines need to float high), which may be sufficient for slower buses. However, external pull-ups are required for reliable operation at higher speeds. Resistor values will typically be 4.7kΩ or 10kΩ.

I²C devices

There are a huge variety of I²C devices available, and the communication patterns they follow are not necessarily all the same. This section gives a high-level overview of the basics of I²C communication, which is relevant to all devices.

Addressing

I²C addressing is how the controller selects which device on the bus it wants to talk to. Since many devices share the same two wires, this aspect is critical. On an I²C bus, all devices see every transmission, but only the device whose address matches will respond; all other devices stay silent.

So, the first byte after the start of an I²C message (START) is always about which device the message is for. Most I²C devices use 7-bit addresses.

Table 5.
I²C addressing

Target address							Read/write bit
A6	A5	A4	A3	A2	A1	A0	0 write to target, 1 read from target

This makes an 8-bit address byte on the wire, but the actual address is still 7 bits.

Warning

The datasheet for some I²C devices will list the combined 8-bit value as the address, in which case it must be shifted right by 1 (or divided by 2) to get the 7 bit address for use under Linux.

Example

For a target address of (binary): 1001000 (0x48), the byte sent would be:
When writing: 10010000 (0x90)
When reading: 10010001 (0x91)

7-bit addresses technically range from 0x00 to 0x7F, but many are reserved; the usable addresses are typically 0x08 to 0x77.

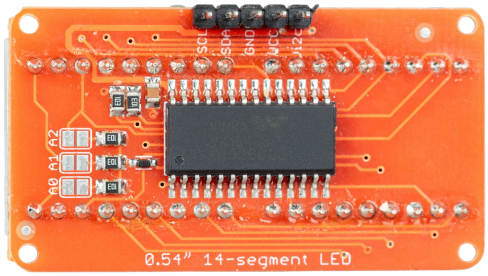
Note

There is also a 10-bit addressing available — Raspberry Pi devices do support it, but it is very rare and not detailed here.

Address pins on the device

Many I²C chips and breakout boards have a number of pins often labelled A0, A1, A2, or similar. These let you change part of the address by tying the pins to ground (for a 0) or VCC (for a 1).

Figure 4.
On the pictured device, 0 Ohm resistors or solder blobs are used to set each bit. Three bits means up to eight devices of the same type can be used on the same bus



Example

A device or chip may have a base address of 0x50, with a binary number defined by three extra pins; this gives us eight unique addresses from 0x50 to 0x57. This allows us to have up to eight similar devices on the same bus, each with a different address.

> **Note**
>
> Check the datasheet to determine the exact use of address pins for your device, as this can vary.

Address matching

After the controller sends the address byte, it looks for a response from the device; no response means no device with that address was present. Controllers can scan the bus using this mechanism; see Using `i2cdetect` to detect I²C buses and scan for connected devices later in this document.

More detail on data transmission

While in-depth knowledge of this section is not vital when using I²C, it is always useful to have an understanding of the underlying concepts.

Basic transmission flow

A complete I²C transfer looks like the following:

START	Address+R/W	ACK	Data	ACK	...	STOP

START and STOP conditions

The START and STOP conditions frame the transmission:

START SDA goes from HIGH to LOW while SCL is HIGH
STOP SDA goes from LOW to HIGH while SCL is HIGH

A repeated START can be used instead of STOP to keep control of the bus.

Sending a data byte

Each data byte has 8 bits, with the most significant bit (MSB) sent first. For each bit:

1. The controller sets SDA
2. The controller toggles SCL to HIGH
3. The receiver samples SDA while SCL is HIGH
4. SCL goes LOW → next bit

After 8 bits, the transmitter releases SDA.

ACK/NACK bit (ninth clock)

The ninth clock pulse is special, as it confirms successful reception and controls data flow.

- ACK (0), receiver pulls SDA LOW
- NACK (1), SDA remains HIGH

Write operations (controller to target)

Typical write sequence:

1. START
2. Target address + W
3. Target ACK
4. Data byte 1
5. Target ACK
6. Data byte 2
7. Target ACK

8. STOP

Read operations (target to controller)

Typical read sequence:

1. START
2. Target address + R
3. Target ACK
4. Target sends data byte 1
5. Controller ACK
6. Target sends data byte 2
7. Controller NACK (signals last byte)
8. STOP

The controller sends NACK on the final byte to end the read.

Combined write-reads (very common)

Used when reading registers. For example:

1. START
2. Address + W
3. ACK
4. Register address
5. ACK
6. Repeated START
7. Address + R
8. ACK
9. Data byte(s)
10. NACK
11. STOP

This keeps the bus and the target state intact.

Clock stretching

If the targets are slow at processing data, or are waiting for the data to be ready, they may hold SCL LOW to delay the controller. The controller must wait until SCL is released.

Warning

On Raspberry Pi devices prior to Raspberry Pi 5, the peripheral-based clock stretching does not work correctly. However, bit-banged I²C does.

Bus states during data transfer

- SDA changes only when SCL is LOW
- SDA is stable when SCL is HIGH

Violating this causes corrupted data.

Error handling

Missing ACK Device not ready or has the wrong address
Arbitration lost Multi-controller conflict
Timeouts Bus is stuck LOW

More detail on repeated START

An I²C repeated START enables the master to start a new transfer without releasing the bus. A repeated START is the same electrical condition as a normal START, but it happens without a preceding STOP.

- SDA goes HIGH to LOW while SCL is HIGH

- The bus is still considered busy
- The same master keeps control of the bus

Repeated STARTs allow the controller to:

- Change direction (write to read)
- Change address
- Prevent other masters from taking the bus
- Keep the target's internal state intact

Useful I²C Linux commands

These commands are all supplied by the `i2c-tools` package, which is installed by default on Raspberry Pi OS. Please use the Linux man system for full information on each of these commands (e.g. `man ic2dump` in a terminal window).

Using `i2cdetect` to detect I²C buses and scan for connected devices

The `i2cdetect` command is a user-space tool employed in Linux to scan I²C buses for connected devices and produce a table of active device addresses. It is essential for troubleshooting and verifying connections.

Note

i2cdetect uses different mechanisms to probe the device based on the address. 0x30–0x37 and 0x50–0x5f use SMBus Read; all others use SMBus Quick.

Common `i2cdetect` usage

```
# List all I2C buses:
i2cdetect -l
# Scan a specific bus (e.g. bus 1):
sudo i2cdetect -y 1
# Scan with read-byte commands (more effective):
sudo i2cdetect -r -y 1
```

Understanding the output matrix

When running `i2cdetect`, the output provides a matrix of hex addresses (0x00–0x7F):

Table 6.
`i2cdetect` table output

Result	Meaning
--	No device is detected at this address
UU	The address is in use by a kernel driver and cannot be probed unless the force flag is used. Caution should be exercised over the use of this flag as it can result in the device and kernel driver becoming out of sync.
2d (or any hex code)	A device is present and responded to the probe at this address

When first connecting an I²C device to a Raspberry Pi SBC, using `i2cdetect` is essential to determining whether the device has been detected on the I²C bus. If an address appears in the matrix, the device has been found.

Note

I²C does not support enumeration. During kernel boot, if I²C is enabled, the kernel will attempt to instantiate drivers for all devices referenced in the device tree. These drivers may attempt to determine whether the device is actually present, but this is not guaranteed. Devices that are handled by kernel drivers should not be accessed from user space, except via the driver's I/O controls (IOCTLs/`ioctl`), as this may cause data corruption. If a device is plugged in after kernel boot (which is not recommended, as I²C is not designed for installation whilst the power is on), then it is possible to load a driver using `dtoverlay` on the command line.

Using `i2cdump` to display a device's register content

`i2cdump` is a command-line tool in Linux that examines the registers of I²C devices. It is often used for debugging, identifying sensors, or reading EEPROM contents.

Common `i2cdump` usage

```
# Dump all 256 registers of an I2C device at address 0x50 on bus 1 without prompting for confirmation (useful
for scripts):
sudo i2cdump -y 1 0x50
# Dump only registers 0x00 through 0x3f from device 0x2d on bus 1:
sudo i2cdump -r 0x00-0x3f 1 0x2d
```

Using `i2cset` to write values to device registers

The `i2cset` command is a powerful, yet potentially dangerous, utility used to set registers on I²C devices from the command line.

Example `i2cset` usage

```
# Write value 0x42 to the 8-bit register 0x11 of a device at the 7-bit address 0x2d on I2C bus 1; the -y flag skips
user confirmation, which is useful for scripting:
i2cset -y 1 0x2d 0x11 0x42
```

For more complex operations, `i2ctransfer` is a better option.

Using `i2cget` to read from device registers

The `i2cget` command is used to read values from I²C device registers.

Common `i2cget` usage

```
# Read register 0x11 from a device with 7-bit address 0x2d on bus 1:
i2cget -y 1 0x2d 0x11
# Read a single byte from device 0x48 on bus 1 without specifying a register address (uses the current or default
register, common on SMBus):
i2cget -y 1 0x48
# Block read, then read 8 bytes starting from register 0x00 on a device at 0x50 on bus 4:
i2cget -y 4 0x50 0x00 i 8
# EEPROM read — set the internal pointer register of an EEPROM (example address: 0x50 on bus 9) to 0x00, then read 2
bytes, relying on auto-increment:
i2cset -y 9 0x50 0x00 ; i2cget -y 9 0x50 ; i2cget -y 9 0x50
```

Warning

`i2cdump`, `i2cset`, and `i2cget` are all SMBus tools. While they are appropriate for some devices, using `i2ctransfer` is generally the best option.

Using `i2ctransfer` to perform compound I²C transfers

`i2ctransfer` is a powerful command-line tool used to send combined I²C messages (write + read, write + write, etc.) in a single, uninterrupted transaction. This is crucial for devices that require a repeated START condition to function properly, such as sensors, EEPROMs, and LCD displays.

Reading data from a device (write + read)

To read data, you often need to send a write message to set the register address, followed by a read message.

```
# On bus 0, write 1 byte (0x64) to device 0x50, then read 8 bytes:
i2ctransfer -y 0 w1@0x50 0x64 r8
```

Writing data to a device

```
# On bus 0, write 3 bytes to device 0x50; the first byte (0x42) is the register address, followed by data 0xff
and 0xfe:
i2ctransfer -y 0 w3@0x50 0x42 0xff 0xfe
```

Writing multiple bytes using auto-increment

```
# Write 17 bytes to device 0x50, starting at register 0x42; the data starts with 0xff and decreases by 1 for each
subsequent byte (0xff, 0xfe, 0xfd...):
i2ctransfer -y 0 w17@0x50 0x42 0xff-
```

Complex combined transfers (write + write + read)

```
# On bus 1, perform a complex transaction: write 2 bytes to register 0x8f, write 1 byte to register 0x0f, then read
2 bytes from device 0x5f:
i2ctransfer -y 1 w2@0x5f 0x8f 0x01 w1@0x5f 0x0f r2
```

Software support

Language bindings

C/C++ (direct access)

You can talk to I²C directly via:

- `/dev/i2c-X`
- `ioctl()` calls (I2C_SLAVE, I2C_RDWR, etc.)

Example:

```
fd = open("/dev/i2c-1", ...)
ioctl(fd, I2C_SLAVE, addr)
read(fd,...)/write(fd, ...)
```

This is fast, flexible, and very common in embedded Linux.

Alternatively, you can use libraries such as `libi2c`, which is a user-space library provided by the `i2c-tools` package that allows C/C++ programs to communicate with I²C and SMBus devices on Linux. It wraps `ioctl` calls to the `/dev/i2c-dev` kernel driver, offering functions for reading/writing data, such as `i2c_smbus_write_byte_data`. It requires linking with `-li2c`.

Python

There are many options in Python, depending on the platform. Some examples include:

- **smbus/smbus2** This is the most commonly used library
- **python-periphery** Clean API, embedded-friendly
- **Adafruit Blinka** Higher-level device abstractions

Example:

```
from smbus2 import SMBus
with SMBus(1) as bus:
    bus.write_byte_data(0x48, 0x00, 0xFF)
```

Example

C

This example uses raw IOCTLs to send and receive data in one transaction. Although there is a C library (libi2c) available, it is limited to SMBus transactions, which do not support repeated START.

The following shows the IOCTLs:

- Opening `/dev/i2c-1`
- Talking to an I²C target at address 0x50
- Writing one register address
- Immediately reading back two bytes from that register using a repeated START (no STOP in between)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>
#include <linux/i2c.h>

int main(void)
{
    int fd;
    const char *device = "/dev/i2c-1";
    uint8_t slave_addr = 0x50;

    /* Buffers */
    uint8_t reg = 0x10;        // Register to read from
    uint8_t data[2] = {0};

    /* I2C messages */
    struct i2c_msg messages[2];
    struct i2c_rdwr_ioctl_data ioctl_data;

    /* Open I2C device */
    fd = open(device, O_RDWR);
    if (fd < 0) {
        perror("Failed to open I2C device");
        return 1;
    }

    /* Message 0: Write register address */
    messages[0].addr  = slave_addr;
    messages[0].flags = 0;              // Write
    messages[0].len   = 1;
    messages[0].buf   = &reg;

    /* Message 1: Read data */
    messages[1].addr  = slave_addr;
    messages[1].flags = I2C_M_RD;        // Read
    messages[1].len   = sizeof(data);
    messages[1].buf   = data;

    /* ioctl data */
    ioctl_data.msgs  = messages;
    ioctl_data.nmsgs = 2;

    /* Perform combined I2C transaction on the file descriptor returned by open */
    if (ioctl(fd, I2C_RDWR, &ioctl_data) < 0) {
```

```
        perror("I2C_RDWR ioctl failed");
        close(fd);
        return 1;
    }

    printf("Read data: 0x%02X 0x%02X\n", data[0], data[1]);

    close(fd);
    return 0;
}
```

Conclusion

The I²C protocol remains one of the most practical and efficient ways to connect peripherals to Raspberry Pi SBCs. With just two signal lines — SDA and SCL — you can interface with a wide range of sensors, displays, ADCs, DACs, RTCs, and expansion modules while keeping wiring simple and scalable.

This white paper has explored how to enable I²C on Raspberry Pi computers, identify connected devices, and communicate with them using tools and libraries available in Raspberry Pi OS. With this foundation, you can confidently build reliable I²C-based projects.

Whether you're logging environmental data, driving an OLED display, or integrating multiple sensors into a larger system, I²C provides a flexible and robust communication backbone. As your projects grow in complexity, mastering I²C will allow you to expand your Raspberry Pi's capabilities with minimal hardware overhead and maximum efficiency.

Once the basics are in place, the next step is experimentation — connect a new device, explore its datasheet, and start building. The Raspberry Pi ecosystem offers nearly endless I²C-compatible components, and each one presents an opportunity to turn ideas into working prototypes.

Contact us for more information

Please contact applications@raspberrypi.com if you have any queries about this whitepaper.

Web: www.raspberrypi.com

Raspberry Pi

Raspberry Pi Ltd