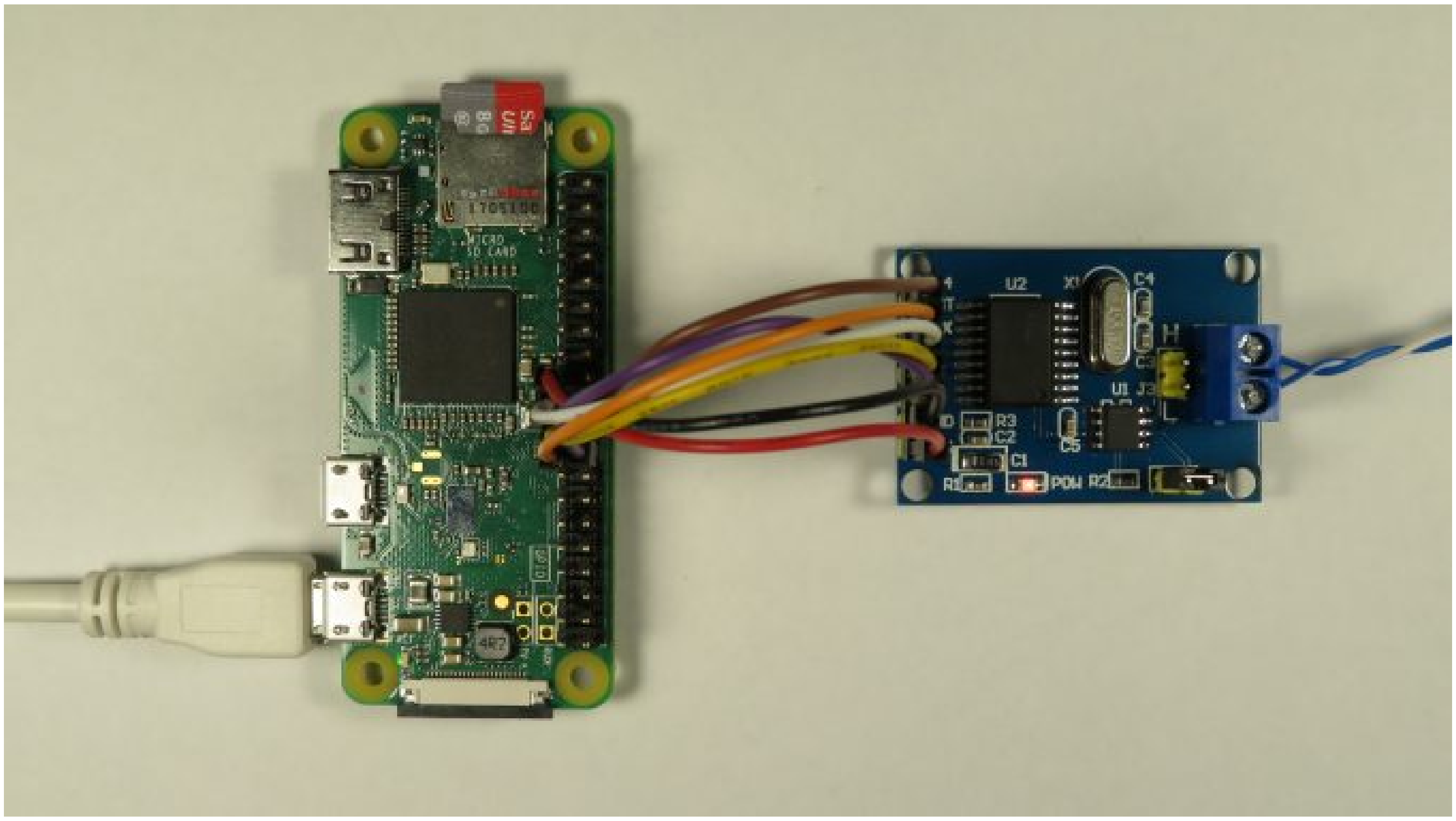
Adding CAN to the Raspberry PI



The CAN bus (Controller Area Network) was originally designed by **Bosch** for the automotive market to connect ECUs (Engine/Electronic Control Units) together. Today, this robust communications bus is commonly found, not only in vehicles, but also on the factory floor in automation (e.g. **CANOpen**) and other applications such as PV solar inverter/battery Energy Storage Systems (ESS).

The Raspberry PI doesn't natively support CAN. The Broadcom SoCs (System on a Chip) used by the Raspberry PI doesn't include a CAN controller.

The Linux kernel supports CAN and includes SocketCAN drivers for the **Microchip MCP2515 Stand-alone CAN Controller with SPI Interface**. Various add-on expansion boards ('hats') exist for the Raspberry PI including:

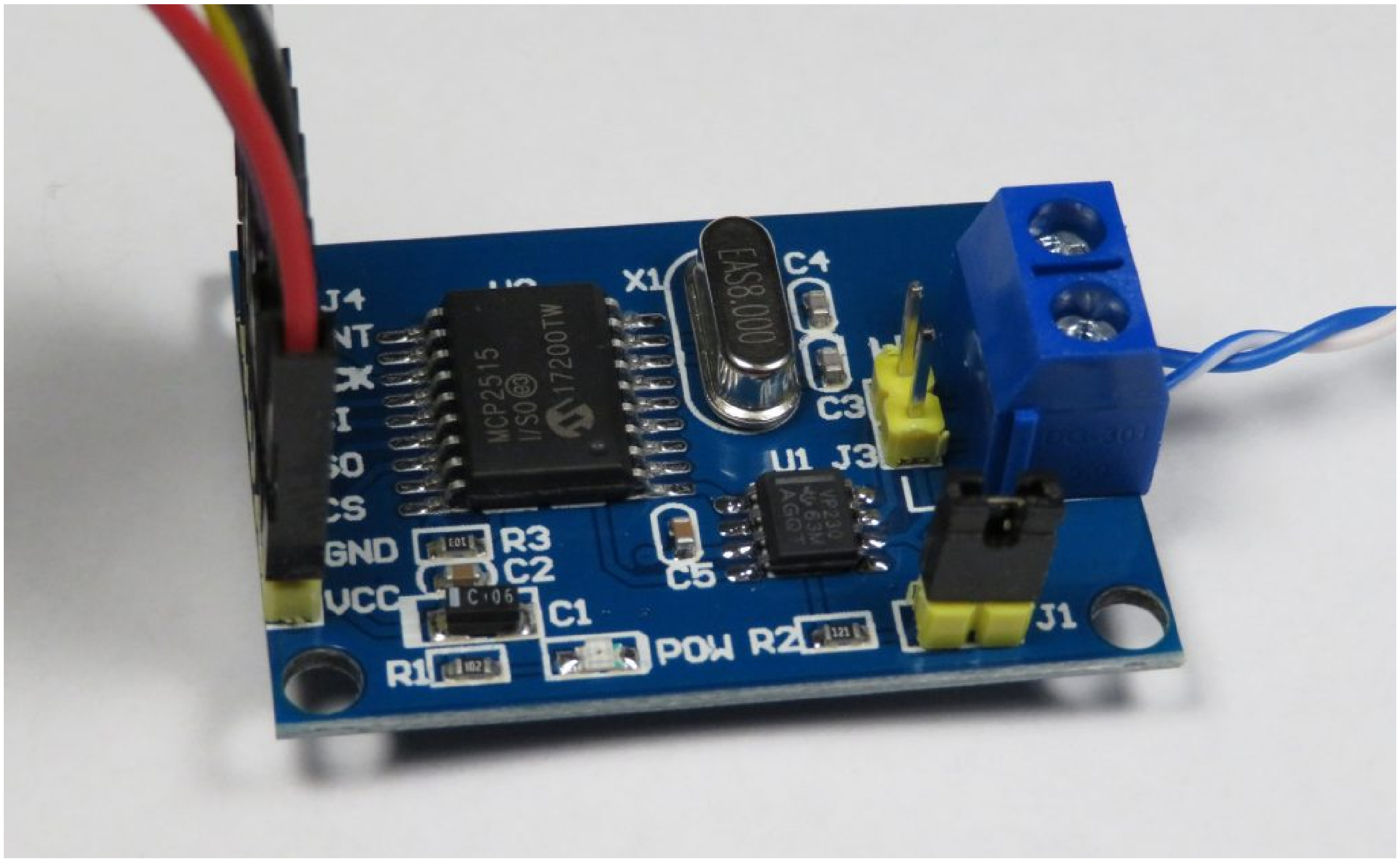
- **PiCAN2**, **PiCAN3** and other variants from **SK Pang Electronics**
- **RS485 CAN HAT** from **Waveshare International Limited**
- **2-Channel Isolated CAN FD Expansion Hat** from Waveshare
- MCP2515 CAN Bus Module Board TJA1050 from **Aliexpress**, **Banggood** and ebay.

MCP2515 CAN Bus Module Board with ~~TJA1050~~ SN65HVD230

While this module is dirt cheap and extremely prevalent, it is not 3.3V compatible and hence Raspberry PI compatible. This board is designed to work from 5V only.

The on-board Microchip MCP2515 CAN Controller supports a wide voltage range from 2.7 to 5.5V. However, the CAN Transceiver, the **TJA1050** from **NXP** only supports 4.75 to 5.25V.

One solution would be to power the board from 3.3V and cut the tracks on the PCB to power the TJA1050 from 5V. If you want to take this path, you can find **instructions here**.

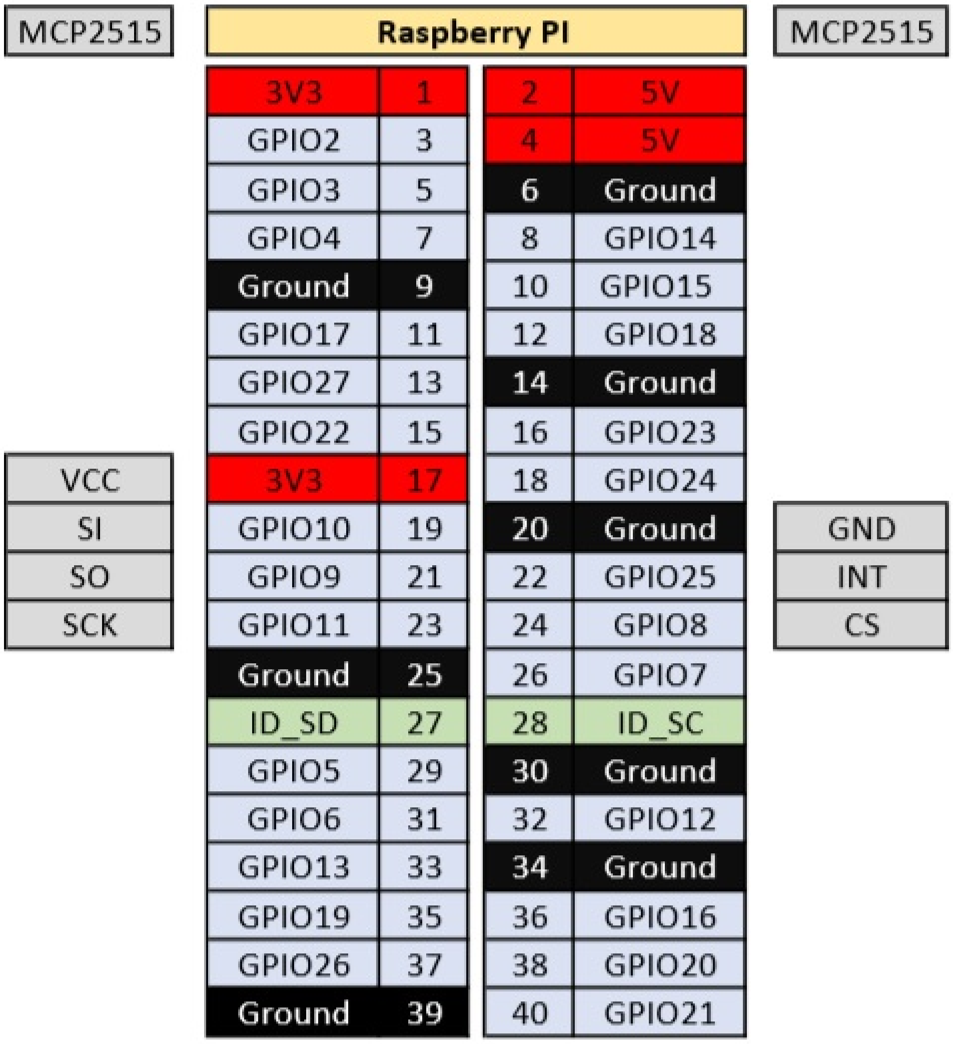


U1 has been replaced with a 3.3V CAN Transceiver – SN65HVD230

Alternatively, if you are proficient at SMD rework, you can remove the 5V CAN Transceiver and install a 3.3V version. I did this and opted to replace the 5V TJA1050 with the 3.3V **SN65HVD230**.

Wiring

Below is the recommended wiring configuration to connect the MCP2515 CAN Bus Module Board to the Raspberry PI's 40 pin connector.



MCP2515	Raspberry PI				MCP2515
	3V3	1	2	5V	
	GPIO2	3	4	5V	
	GPIO3	5	6	Ground	
	GPIO4	7	8	GPIO14	
	Ground	9	10	GPIO15	
	GPIO17	11	12	GPIO18	
	GPIO27	13	14	Ground	
	GPIO22	15	16	GPIO23	
VCC	3V3	17	18	GPIO24	
SI	GPIO10	19	20	Ground	GND
SO	GPIO9	21	22	GPIO25	INT
SCK	GPIO11	23	24	GPIO8	CS
	Ground	25	26	GPIO7	
	ID_SD	27	28	ID_SC	
	GPIO5	29	30	Ground	
	GPIO6	31	32	GPIO12	
	GPIO13	33	34	Ground	
	GPIO19	35	36	GPIO16	
	GPIO26	37	38	GPIO20	
	Ground	39	40	GPIO21	

Driver Installation

Without an **ID EEPROM** on the 'hat' specifying the hardware, the Linux kernel will not automatically discover the CAN Controller on the **SPI interface**. To load the appropriate driver, you must specify device tree overlay settings at boot.

Add the following line to your /boot/config.txt file:

```
dtoverlay=mcp2515-can0,oscillator=8000000,interrupt=25
```

The oscillator parameter should be set to the actual crystal frequency found on your MCP2515. This frequency can change between modules, and is commonly either 16 or 8 MHz. My MCP2515 CAN Bus module board has a 8MHz on-board crystal and hence, I set the above line to 8000000.

The interrupt parameter specifies the Raspberry PI GPIO pin number. We have connected the INT pin to GPIO25.

By default, the mcp2515 driver uses a maximum SPI frequency of 10MHz (as per the MCP2515 datasheet). You can also specify the overlay an optional parameter spimaxfrequency, e.g. spimaxfrequency=2000000 to slow down the SPI clock to help with signal integrity issues – e.g. if you have flying leads, rather than a PCB!

Note: Historic documentation may suggest adding dtparam=spi=on and dtoverlay=spi-bcm2835-overlay. While the SPI master is not enabled by default, and specifying dtparam=spi=on will enable it, so will the mcp2515-can0 overlay. The spi-bcm2835-overlay was used to specify a newer bcm2835 SPI driver (vs the older bcm2708), but since version 4.4 of the kernel, bcm2835 is now the default driver.

Now reboot your PI. You should see the following kernel messages on boot:

```
[   20.248951] CAN device driver interface
[   20.499256] mcp251x spi0.0 can0: MCP2515 successfully initialized.
```

Naturally, if you see "*Cannot initialize MCP2515. Wrong wiring?*", check the power and wiring of your CAN controller.

Once you have rebooted and the driver has been successfully loaded, you can manually bring up the CAN interface using:

```
 sudo /sbin/ip link set can0 up type can bitrate 500000
```

The bitrate here is specified as 500kHz, but this may be changed to suit other devices on your CAN bus.

To automatically bring up the interface on boot, edit your /etc/network/interfaces file and add the following:

```
auto can0
iface can0 inet manual
    pre-up /sbin/ip link set can0 type can bitrate 500000 triple-sampling on
restart-ms 100
    up /sbin/ifconfig can0 up
    down /sbin/ifconfig can0 down
```

The SocketCAN interface behaves just like a network interface. You should be able to get various statistics using *ifconfig* (interface configuration)

```
pi@raspberrypi:~ $ ifconfig

can0: flags=193  mtu 16

        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 10
(UNSPEC)

        RX packets 0  bytes 0 (0.0 B)

        RX errors 0  dropped 0  overruns 0  frame 0

        TX packets 0  bytes 0 (0.0 B)

        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

CAN-utils

CAN-utils is a collection of extremely useful debugging tools using the SocketCAN interface. It includes applications such as:

- **candump** – Dump can packets – display, filter and log to disk.
- **canplayer** – Replay CAN log files.
- **cansend** – Send a single frame.
- **cangen** – Generate random traffic.
- **canbusload** – display the current CAN bus utilisation.

CAN-utils source can be obtained from the GitHub repository: **https://github.com/linux-can/can-utils**

Alternatively, pre-compiled binaries can be installed using:

```
sudo apt-get install can-utils
```

Code Development

Check out our tutorial for **example C SocketCAN code**.

Performance

On buses which high frame rates, the mcp251x driver may struggle. It is not unusual to see the *irq/160-mcp251x* kernel process using 25 percent of the CPU and the *spi0* kernel using another 10 percent.

Enabling DMA doesn't seem to make a material difference.

For most applications, it is unlikely you will want to capture and process every CAN frame. More likely than not, you are only only after frames sent to a specific address or range of addresses. This is how CAN buses were envisioned.

Given any other CAN device, one would just set a CAN filter and mask. This would filter out any frames not of interest and leave the firmware/software to deal with the important frames.

While this can be done, one of the deficiencies of the SocketCAN interface is that this filtering happens at a kernel module level. The filter and mask is never passed to the mcp251x for hardware level filtering. As a result, high CPU utilisation is still present when using filtering.

mcp251x.ko driver with hardware level filtering

One solution would be to modify the mcp251x driver to enable hardware filtering. Currently no means of passing down the filter or mask exists, but it could be either hardcoded, or loaded in at module load.

If this is of interest, see **https://github.com/craigpeacock/mcp251x**

*If high frame rates are required, you may be better off to switch to alternative embedded Linux platforms such as the **BeagleBone**. The BeagleBone has internal CAN Controllers and support high frame rates with ease.*