

07: Raspberry Pi, Memory

Microcontrollers

Stefan Huber

Dept. for Information Technologies and Digitalisation
FH Salzburg

Summer 2026

Raspberry Pi

Raspberry Pi

Data reported by commands `lscpu`¹ and `pinout`:

- ▶ BCM2837 SoC
- ▶ Quad core CPU (Cortex A53, ARMv8-A) at 1.2 GHz and 1 GiB RAM
- ▶ 1 Ethernet, 4 USB 2.0, WiFi, Bluetooth, ...

Use `pinout` to locate the GPIO pins on the pin rows.



J8:			
3V3	(1)	(2)	5V
GPIO2	(3)	(4)	5V
GPIO3	(5)	(6)	GND
GPIO4	(7)	(8)	GPIO14
GND	(9)	(10)	GPIO15
GPIO17	(11)	(12)	GPIO18
GPIO27	(13)	(14)	GND
GPIO22	(15)	(16)	GPIO23
3V3	(17)	(18)	GPIO24
GPIO10	(19)	(20)	GND
GPIO9	(21)	(22)	GPIO25
GPIO11	(23)	(24)	GPIO8
GND	(25)	(26)	GPIO7
GPIO0	(27)	(28)	GPIO1
GPIO5	(29)	(30)	GND
GPIO6	(31)	(32)	GPIO12
GPIO13	(33)	(34)	GND
GPIO19	(35)	(36)	GPIO16
GPIO26	(37)	(38)	GPIO20
GND	(39)	(40)	GPIO21

¹ In 32-bit mode the CPU architecture is reported as armv7l.

BCM2835 peripherals

SoC comes with various peripherals:

- ▶ GPIO
- ▶ Timers, PWM
- ▶ Interrupt controller, DMA²controller
- ▶ UART, SPI, I²C, USB

SoCs are all based on BCM2835³:

BCM2835	A, B, B+
BCM2836	2B
BCM2837	3, later 2
BCM2837B0	3A+, 3B+
BCM2711	4B

Bit banging

Directly setting GPIO pins to create transmission signals directly rather than using dedicated hardware. For instance, SPI can be implemented via bit banging on four GPIO pins.

² Direct memory access (DMA) allows peripherals to directly access RAM to exchange data with the CPU.

³ See overview at [\[rpi-BCM2837\]](#)

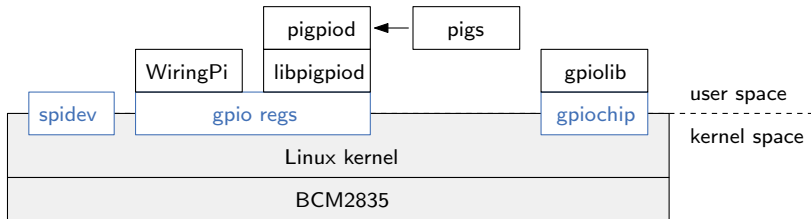
(Not so good) documentation:

- ▶ *BCM2835 ARM Peripherals*. Broadcom Corporation. 2012. URL: <https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf>
- ▶ Errata: *BCM2835 datasheet errata*. URL: https://elinux.org/BCM2835_datasheet_errata

Linux kernel interfacing

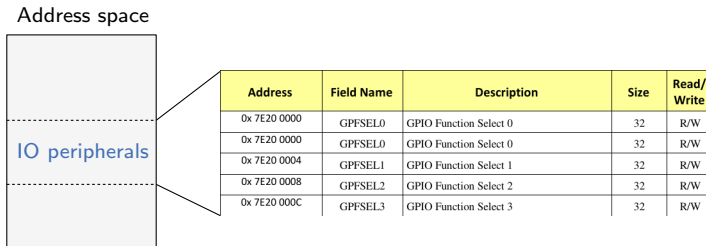
There are different ways to interface with the Linux kernel:

- ▶ Direct BCM2835 register access **in memory**, e.g., for GPIO. Additional user-space libraries make this more convenient, like the WiringPi.
- ▶ Dedicated interfaces for dedicated hardware classes, e.g., spidev for SPI or gpiochip for GPIO.



Raspberry Pi I/O peripherals via memory mapping

The I/O peripherals are accessed through registers mapped in memory.⁴



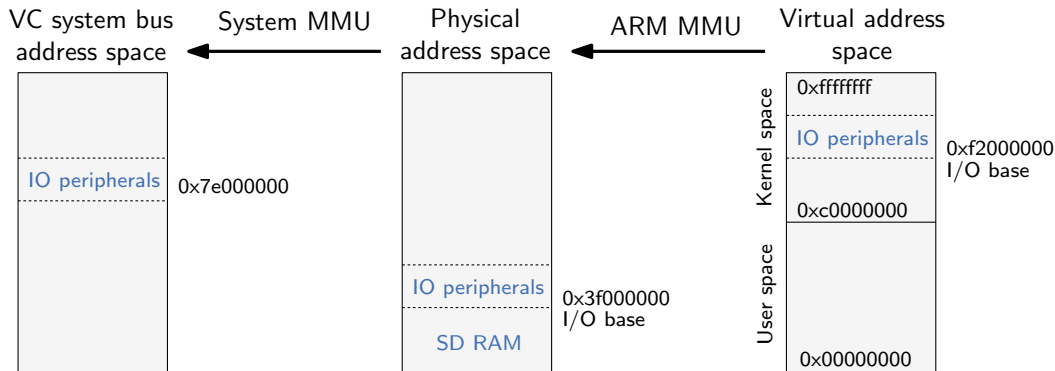
The **IO peripherals** memory region has to be mapped into the **virtual memory address space** of each Linux process.

⁴ BCM2835, p. 4.

From system bus to virtual memory

The BCM2835 SoC has **two MMUs** for a two-level memory mapping:

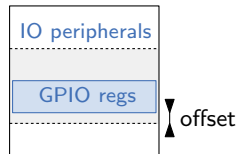
- ▶ The ARM processor is kind of a co-processor to the GPU in the *VideoCore* architecture.
- ▶ A system bus address `0x7eNNNNNN` is mapped to the virtual address `0xf2NNNNNN` in kernel space.



Accessing GPIO registers

The GPIO registers start at system bus address `0x7e200000`:

- ▶ Hence, at offset `0x00200000` within IO peripherals.
- ▶ Hence, they start at `0x3f200000` in physical memory.
- ▶ Hence, they start at `0xf2200000` in virtual memory, which is in kernel space.



Two possibilities to access GPIO registers:

- 1 The character device file `/dev/mem` contains an image of the physical memory. Only user root and group `kmem` can access it. Security!
- 2 The character device file `/dev/gpiomem` contains only the GPIO memory region. All users in the group `gpio` can access it.

```
1 $ ls -l /dev/gpiomem /dev/mem
2 crw-rw---- 1 root gpio 247, 0 Aug 20 11:17 /dev/gpiomem
3 crw-r----- 1 root kmem 1, 1 Aug 20 11:17 /dev/mem
```

GPIO handling: Accessing registers

- ▶ We open the character device file `/dev/gpiomem`.
- ▶ We map the file/device into memory via `mmap()`. See the man page: `man mmap`

```
1  int gpiofd = open("/dev/gpiomem", O_RDWR);
2  if (gpiofd < 0) {
3      perror("Cannot open /dev/gpiomem");
4      exit(EXIT_FAILURE);
5  }
6
7  void* gpiomap = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, gpiofd, 0);
8  if (gpiomap == MAP_FAILED) {
9      perror("Cannot mmap gpio opened from /dev/gpiomem");
10     exit(EXIT_FAILURE);
11 }
12
13  /* Now we can operate on gpiomap memory... */
```

Digital output via GPIO

[BCM2835, p. 90] describes the following registers:

- ▶ GPIO function select register, GPIO pin output set register, GPIO pin output clear register.

This example sets GPIO 13 high for 200 ms and clears it again:

```
1  /* GPIO 13 is controlled by function select register 1, at offset 0x4. */
2  uint32_t* const fselreg1 = (uint32_t*) (gpiomap + 0x04);
3  /* Bits 9-11 in function select register 1 determine the function of pin
4   * 13. The bits 001 mean output pin.*/
5  *fselreg1 &= ~(0x7 << 9);
6  *fselreg1 |= (1 << 9);
7
8  /* Setting a bit in output clear register sets the pin low. Setting a bit
9   * in the output set register sets the pin high. */
10 uint32_t* const outputset0 = (uint32_t*) (gpiomap + 0x1c);
11 uint32_t* const outputclr0 = (uint32_t*) (gpiomap + 0x28);
12 *outputset0 |= (1 << 13);
13 usleep(200000);
14 *outputclr0 |= (1 << 13);
```

Pin modes

A pin has configurable pull-up and pull-down resistors.⁵

A pin has alternative functions:

- ▶ The function is chosen by the function selection registers.⁶
- ▶ We can select for each pin between modes input, output or one of 2–6 alternative functions.

	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5	<reserved>			
GPIO1	High	SCL0	SA4	<reserved>			
GPIO2	High	SDA1	SA3	<reserved>			
GPIO3	High	SCL1	SA2	<reserved>			
GPIO4	High	GPCLK0	SA1	<reserved>			ARM_TDI
GPIO5	High	GPCLK1	SA0	<reserved>			ARM_TDO
GPIO6	High	GPCLK2	SD_E_N / SE	<reserved>			ARM_RTCK
GPIO7	High	SPI0_CE1_N	SWE_N / SRW_N	<reserved>			
GPIO8	High	SPI0_CE0_N	SD0	<reserved>			
GPIO9	Low	SPI0_MISO	SD1	<reserved>			
GPIO10	Low	SPI0_MOSI	SD2	<reserved>			
GPIO11	Low	SPI0_SCLK	SD3	<reserved>			
GPIO12	Low	PWM0	SD4	<reserved>			ARM_TMS
GPIO13	Low	PWM1	SD5	<reserved>			ARM_TCK

Figure: Top part of an overview table of alternative functions, see [BCM2835, p. 102].

⁵ BCM2835, p. 100.

⁶ BCM2835, p. 91.

The Wiring Pi as HAL hides details of the Raspberry Pi versions:

- ▶ For instance, prior Model 2, which had less RAM, the I/O base address in physical memory was 0x20000000, but changed to 0x3f000000.
- ▶ Different number of GPIO pins, different functionalities.

SPI on the Raspberry Pi

The Linux kernel supports SPI, not only for the Raspberry Pi.

- ▶ The header `linux/spi/spidev.h` exposes the API to the `spidev` driver.
- ▶ Character devices `/dev/spidev*` are accessible to group `spi`.

```
1 $ ls -l /dev/spidev*  
2 crw-rw---- 1 root spi 153, 0 Aug 20 11:17 /dev/spidev0.0  
3 crw-rw---- 1 root spi 153, 1 Aug 20 11:17 /dev/spidev0.1
```

- ▶ `spidevX.Y` means SPI device `Y` on bus `X`.

The tool `spidev_test` can be used to make a loopback test by short-cutting MOSI with MISO.

- ▶ Download its source code to find out how to use the `spidev` API. [[spidevtest](#)]

A glimpse into the spidev API (init)

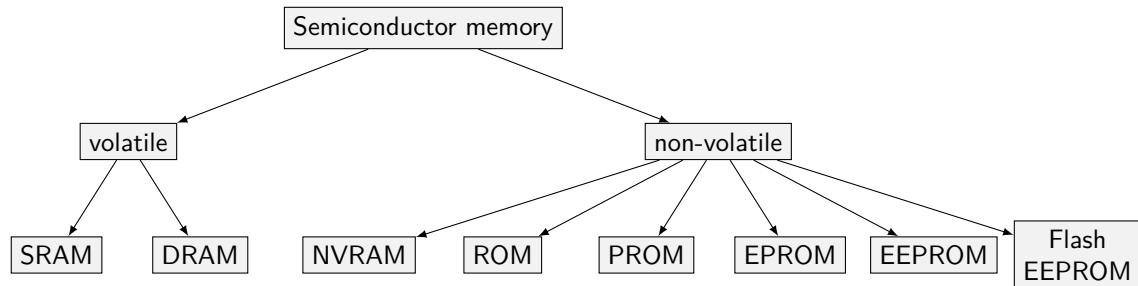
```
1 const uint8_t bits = 8;
2 const uint32_t speed = 100000;
3
4 /** Open the first spidev device, i.e., /dev/spidev0.0 and return its file
5 * descriptor. After open() the device is initialized to 8-bit words and 100 kHz
6 * speed. */
7 static int open_spi() {
8     int ret;
9     int spifd = open("/dev/spidev0.0", O_RDWR);
10    if (spifd < 0)
11        perror("Cannot open /dev/spidev0.0");
12
13    ret = ioctl(spifd, SPI_IOC_WR_BITS_PER_WORD, &bits);
14    if (ret < 0)
15        perror("Cannot set SPI word size");
16    ret = ioctl(spifd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
17    if (ret < 0)
18        perror("Cannot set SPI max speed");
19
20    return spifd;
21 }
```

A glimpse into the spidev API (transceive)

```
1 /* Transceive a byte over SPI device at file descriptor spifd. That is,  
2 * transfer len bytes of the array tx and receive as many bytes and store them  
3 * in rx. See open_spi() to open an spidev device. */  
4 static int send_spi_message(int spifd, uint8_t* tx, uint8_t* rx, size_t len) {  
5     struct spi_ioc_transfer tr = {  
6         .tx_buf = (unsigned long) tx,  
7         .rx_buf = (unsigned long) rx,  
8         .len = len,  
9         .delay_usecs = 0,  
10        .speed_hz = speed,  
11        .bits_per_word = bits,  
12    };  
13  
14    return ioctl(spifd, SPI_IOC_MESSAGE(1), &tr);  
15 }
```


Memory

Memory technologies



Volatility⁷:

- ▶ Volatile memory loses its data when power is switched off.
- ▶ Non-volatile memory keeps its data after power is switched off.

⁷ Dt. Flüchtigkeit, Unbeständigkeit

Memory types by function

We can distinguish between three types of memory with respect to their function:

Register file The registers of a CPU form a small, temporary “short-term memory” of the CPU.

Data memory Can be volatile and non-volatile. Non-volatile data are e.g. stored configuration parameters.

The volatile data memory contains the stack, the heap, and so on. Volatile data might be short-lived (e.g., stack) or valid as long as the CPU runs (e.g., global variables).

Instruction memory Non-volatile memory for program. In regular operation it is used read-only. It is often quite large even for smaller microcontrollers.

Volatile memory

Why not use non-volatile memory everywhere?

- ▶ Typically volatile memory is *much* faster. Access time of RAM in a PC is in nano seconds range.

RAM:

- ▶ Historically there were memory types that could only be accessed sequentially, whereas **Random Access Memory** (RAM) could access memory in any (e.g., random) fashion.
- ▶ Today we still refer to volatile (data) memory as RAM.

Static versus dynamic:

SRAM A static RAM is an array of flip-flops as 1-bit memory cells to store information. Too costly at large capacities, hence no option for desktop PCs.

DRAM A dynamic RAM uses capacitors instead of flip-flops as 1-bit memory cells. This gives higher memory density and storage capacity and is less expensive, but also slower. It requires cyclic refreshing (every 10 ms to 100 ms) due to leakage current and additional DRAM controllers.

Dual-ported RAM

A **dual-ported RAM** is used as communication buffer.

- ▶ The dual-ported RAM provides two access buses and allows (almost) concurrent access.
- ▶ There are also multi-ported RAMs.

Two or more sub-systems can use it as data exchange buffer:

- ▶ Video RAM (VRAM) between CPU and video/graphics hardware.
- ▶ Data exchange in multi-processor systems.
- ▶ In embedded systems a dual-ported RAM is used for data exchange between sub-systems like the CPU and a co-processor or network devices.

Non-volatile memory

Writing to non-volatile memory is typically much slower and more involved (complicated).⁸

- ROM** Read-Only Memory was historically first non-volatile semiconductor memory. Data is placed by chip manufacturer. Like an ASIC only for mass production.
- PROM** Programmable ROM or one-time programmable (OTP) ROM contain silicon fuses that can be destroyed by high current to encode information.
- EPROM** Erasable PROM can be erased by exposure to ultraviolet light for tens of minutes. An EPROM can be typically erased a few 10 000 times. Information is stored in FETs by applying higher-than-normal voltage to *floating* gates, which lasts for tens of years.
- EEPROM** Electrically EPROM is like an EPROM, except that no UV light is required to clear the FETs but can be done electrically. They can be erased a few 100 000 times.
- Flash** Less expensive than EEPROM, but only entire blocks of memory can be erased (“flashed”). Newer designs can be erased 1 000 000 times. Flash is typically used as program memory in embedded design, memory cards/sticks and SSDs.

⁸ See [ATmega32, p. 18] for ATmega32 EEPROM programming.

A Non-Volatile RAM combines features of RAM with non-volatile memory:

- ▶ Like an SRAM with an internal battery.
- ▶ Or like an SRAM with an EEPROM for storing and restoring data at power off and on.

Universal memory

Storage that aims to combine

- ▶ cost benefit of DRAM
- ▶ speed of SRAM
- ▶ non-volatility of Flash.

Microcontroller memories

Integrated on-chip for microcontrollers:

- ▶ Data memory is typically SRAM for volatile data and some EEPROM or NVRAM for non-volatile data.
- ▶ Program memory is typically some Flash memory.

We **cannot expand** memory for microcontrollers.

- ▶ Choose the right size upfront.
- ▶ Take into account **future features**, **bug fixes** and the like! They may need to fit for the next two decades.
- ▶ Rule of thumb: Leave at least 20 % spare program memory.

Memory access times

The register file is faster than data or instruction memory.

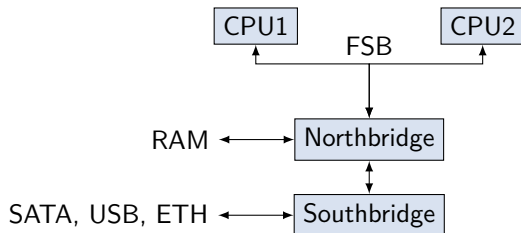
- ▶ Data and instruction memory is external to the CPU and requires bus access.
- ▶ ATmega32: The instructions LD, ST, PUSH, POP operate on data memory and require 2 cycles, but a MOV between registers (even for a word) requires only 1 cycle.

For modern processors DRAM access happens at a clock rate that is, for instance, 10-times slower.

- ▶ FSB clock rate versus CPU clock rate.

Memory access times: Front-side bus

A typical Intel-based northbridge/southbridge⁹ architecture for desktop PCs, see [Ulr07]:



FSB is the front-side bus:

- ▶ It connects the CPUs with RAM.
- ▶ CPU may be clocked at 3.0 GHz, but the FSB may be clocked at 266 MHz.
- ▶ Caches in the CPU reduce the FSB bottleneck.

DMA is direct memory access:

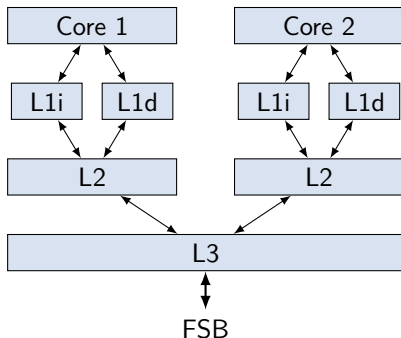
- ▶ Peripherals can write into RAM to exchange data with the CPU.

⁹ Contemporary Intel systems are based on QPI (QuickPath Interconnect), which replaced FSB and is rather a point-to-point routing mechanism with five protocol layers than a "bus".

Memory hierarchy

A typical three-level cache hierarchy for a multi-core CPU:

- ▶ The level 1 cache is separated into instruction (L1i) and data (L1d) cache → Harvard-style to mitigate von Neumann bottleneck
- ▶ Each core has its own L2 cache, but L3 cache is shared.



Typical (vague) values for latency:

- ▶ Register access: Single CPU cycle
- ▶ L1: 1 ns
- ▶ L2: 5 ns
- ▶ L3: 20 ns
- ▶ RAM: 100 ns
- ▶ SSD: 15 000 ns

Write **cache friendly code** to achieve high performance!

Appendix

Raspberry Pi with Python

The Raspberry Pi is well supported by Python:

- ▶ The `gpiozero` Python library provides easy access GPIO pins. It is the Wiring Pi for Python.
- ▶ The `spidev` Python library provides easy access to `spidev` devices.

Employing Python in an embedded system is a more recent cultural disruption.

- ▶ Embedded system development is often done and shaped by hardware-affine engineers, e.g., electrical engineers and control engineers.
- ▶ A personal opinion: Besides the construction of prototypes, demos and proof-of-concepts, a higher-level language like Python reduces the skill threshold and time-to-market, but at costs of performance, temporal determinism and software maintainability¹⁰.

¹⁰

A programming language without a strict type system is often harder to maintain over years in a changing team of multiple persons, because types also express intent, create structure and implicitly document source code.

Python gpiozero demo

The blink demo with `gpiozero`:

```
1 from gpiozero import LED
2 from time import sleep
3
4 led = LED(13)
5 led.on()
6 sleep(0.2)
7 led.off()
```

Documentation:

▶ *gpiozero documentation*. URL: <https://gpiozero.readthedocs.io/en/stable/>

▶ The Python online help:

```
1 python3
2 >>> import gpiozero
3 >>> help(gpiozero)
```

Python spidev demo

```
1 import spidev
2 spi = spidev.SpiDev()
3 # Open /dev/spidev0.0
4 spi.open(0, 0)
5 spi.max_speed_hz = 100000
6
7 # Transfer 3 bytes and store the received 3 bytes in data
8 data = spi.xfer([0x1, 0x2, 0x3])
9 print("Received:", data)
10 spi.close()
```

Documentation:

- ▶ *spidev documentation*. URL: <https://github.com/doceme/py-spidev>
- ▶ The Python online help:

```
1 python3
2 >>> import spidev
3 >>> help(spidev)
```

Accessing GPIO via /dev/mem

```
1 /* RPi 2 and later. Otherwise 0x20000000. */
2 #define IOBASE 0x3f000000
3 #define GPIOBASE (IOBASE + 0x00200000)
4
5 void gpio_access() {
6     int gpiofd = open("/dev/mem", O_RDWR);
7     if (gpiofd < 0) {
8         perror("Cannot open /dev/mem");
9         exit(EXIT_FAILURE);
10    }
11
12    void* gpiomap = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED,
13                        gpiofd, GPIOBASE);
14    if (gpiomap == MAP_FAILED) {
15        perror("Cannot mmap gpio opened from /dev/mem");
16        exit(EXIT_FAILURE);
17    }
18
19    /* Now we can operate on gpiomap memory... */
20 }
```


References I

- [ATmega32] *ATmega32: 8-bit AVR Microcontroller with 32KBytes In-System Programmable Flash.* Atmel Corporation. Feb. 2011.
- [BCM2835] *BCM2835 ARM Peripherals.* Broadcom Corporation. 2012. URL: <https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf>.
- [BCM2835-errata] *BCM2835 datasheet errata.* URL: https://elinux.org/BCM2835_datasheet_errata.
- [gpiozerodoc] *gpiozero documentation.* URL: <https://gpiozero.readthedocs.io/en/stable/>.
- [rpi-BCM2837] *BCM2837 – Raspberry Pi Documentation.* URL: <https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md>.
- [spidevdoc] *spidev documentation.* URL: <https://github.com/doceme/py-spidev>.
- [spidevtest] *Linux tools: spidev_test.* URL: https://github.com/torvalds/linux/blob/master/tools/spi/spidev_test.c.

- [Ulr07] Drepper Ulrich. *What Every Programmer Should Know About Memory*. Tech. rep. Nov. 2007. URL: <https://akkadia.org/drepper/cpumemory.pdf>.