

# Firmware for the BCM2835

## For BCM2835 - Chip Set Aarch64

### Layout, Language, Logic

on July 14, 2025 in Frankfurt Version 1 Revision 1

Archive-ID: BANKO-0001 DOI: 10.xxxx/xxxxx

Duy Nam Schlitz<sup>a\*</sup>

<sup>a</sup> *ÍSAC Sciences and The Namischen Werke, [duynamschlitzresearch@gmail.com](mailto:duynamschlitzresearch@gmail.com)*

<sup>\*</sup> *Corresponding Author*

### Abstract

This document demonstrates the capabilities of the ‘banko’ LaTeX class. It supports multilingual content (CJK), mathematical typesetting, references, structured headings, and customized metadata. This belongs to a collection that also includes Sazuko and the new coming Kougi templates. By the way I am not a Japanese people but I can Japanese. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

|   |          |                                       |          |
|---|----------|---------------------------------------|----------|
| Contents  | 20       | <b>4 Time and Delay</b>               | <b>1</b> |
|   |          | 4.1 NOP . . . . .                     | 1        |
|   | 22       | 4.2 Loop with Subs . . . . .          | 2        |
| <b>2 I Boot and Initializing</b>  | <b>1</b> | 4.3 MMIO Timer . . . . .              | 2        |
|   | 24       | 4.4 cntvct_el0 . . . . .              | 2        |
| <b>1 Compiling the Code</b>   | <b>1</b> | 4.5 Building a System Timer . . . . . | 2        |
| 1.1 Bare Metal Programming . . . . .  | 1        | 4.5.1 System Timer CLO . . . . .      | 2        |
| 1.2 Boot Files . . . . .  | 1        | 4.5.2 ARM Generic Timer . . . . .     | 2        |
| 1.3 Linker File . . . . .   | 1        |                                       |          |
| 1.4 Kernel . . . . .  | 1        | <b>5 Sources</b>                      | <b>3</b> |
| <b>2 MMIO</b>   | <b>1</b> |                                       |          |
| 2.1 UART . . . . .  | 1        |                                       |          |
| 2.2 SPI / I2C . . . . .   | 1        |                                       |          |
| 2.3 Interrupt Controller . . . . .  | 1        |                                       |          |
| 2.4 Set VBAR_EL1 (Exception Table for IRQs,<br>FIQ, SError, SVC etc.) . . . . . | 1        |                                       |          |
| <b>3 GPIO</b>   | <b>1</b> |                                       |          |
| 3.1 Setting GPIO Modes . . . . .  | 1        |                                       |          |
| 3.2 GPSETn and GPCLRn . . . . .   | 1        |                                       |          |
| 3.2.1 Send HIGH to GPIO . . . . .   | 1        |                                       |          |
| 3.2.2 Send LOW to GPIO . . . . .  | 1        |                                       |          |
| 3.3 Read GPIO . . . . .   | 1        |                                       |          |

# I Boot and Initializing

## 1 Compiling the Code

### 1.1 Bare Metal Programming

### 1.2 Boot Files

### 1.3 Linker File

### 1.4 Kernel

## 2 MMIO

### 2.1 UART

### 2.2 SPI/I2C

### 2.3 Interrupt Controller

### 2.4 Set VBAR\_ELI (Exception Table for IRQs, FIQ, SError, SVC etc.)

## 3 GPIO

### 3.1 Setting GPIO Modes

The physical base address of the GPIO is 0x3F200000, which corresponds to 0x7E200000 in the bus address space. The registers from 0 to 49 are divided across 5 registers. The offset for a given pin is calculated as:

$$0x04 \cdot (p_d \bmod 10)$$

To configure a GPIO pin, we assign it one of 8 possible modes. For simple projects, only the Input and Output modes are typically required. The mode is set by modifying the appropriate GPFSELn register. Each pin has a 3-bit field, and its position is calculated as

$$p = 3 \cdot (p_d \bmod 10)$$

We shift the desired mode value to this position and update the register using the |= operator or using a pointer.

The last register is somewhat exceptional: it only contains GPIO pins 50 to 53. However, all other calculations and bit manipulations remain the same.

[1]

<sup>1</sup>We define the selected register as

$$r = p_d \bmod 10$$

## 3.2 GPSETn and GPCLRn

### 3.2.1 Send HIGH to GPIO

The first GPSET register is used to set GPIO pins 0 to 31 to HIGH. The second GPSET register controls GPIO pins 32 to 53, also setting them to HIGH when the corresponding bits are written with 1.

To set a pin to HIGH, write a 1 to the n-th bit of the corresponding GPSET register. All other bits should remain 0. Turning a bit to 0 wouldn't affect the output.

### 3.2.2 Send LOW to GPIO

Just like GPSETn, the GPCLR function is split across two registers. To set a pin to LOW, write a 1 to the corresponding bit position—just as you would with GPSETn. Both registers are write-only, and setting a bit has an immediate effect; reading the register has no meaning.

[1]

## 3.3 Read GPIO

Just like GPSETn and GPCLRn, the GPLEVn (GPIO Pin Level) register is also divided into two parts. Each bit corresponds to the current logic level of one GPIO pin:

- 1 means the pin is HIGH
- 0 means the pin is LOW

Unlike GPSET and GPCLR, which are write-only, GPLEV is read-only. You read the register and check the n-th bit to determine the current state of GPIO pin n.

## 4 Time and Delay

In bare-metal systems, timing is essential for precise GPIO control, communication protocols, and delay handling without relying on operating system services. This section introduces different delay mechanisms for the Raspberry Pi Zero 2 W, from basic NOPs to high-resolution hardware timers.

### 4.1 NOP

A NOP (No Operation) instruction can be used for very short delays. It consumes a single CPU cycle and is typically implemented in ARM with:

```
nop
```

56

However, such delays are imprecise because they:

- depend on CPU clock speed (which may vary),
- are not consistent across architectures or cache states,
- cannot produce exact microsecond/millisecond delays.

They are suitable for very short or loop-compensated waits.

## 4.2 Loop with Subs

}

146

A more stable delay can be implemented with a decrementing loop, using the SUBS instruction and branching:

You can compute nanosecond-precision time via:

148

```
106  mov r0, #100000      @ number of iterations
    delay_loop:
108  subs r0, r0, #1      @ decrement
    bne delay_loop      @ branch if not zero
110
```

$$\text{Time (ns)} = \frac{\text{cntvct\_el0} \times 10^9}{\text{cntfrq\_el0}}$$

While more flexible than NOP, this method is still CPU-speed dependent. Calibration is required to match time units.

## 4.5 Building a System Timer

150

A reusable system timer should abstract the hardware layer. You can build your own module that provides millisecond and microsecond timing with multiple backends.

152

### 4.3 MMIO Timer

The BCM2835/2836/2837 SoC provides a **System Timer** accessible via memory-mapped I/O (MMIO). It counts at 1 MHz and is ideal for microsecond-precision delays.

#### 4.5.1 System Timer CLO

154

The MMIO-based system timer (CLO) is best used for:

- delays in microseconds/milliseconds,
- busy-wait loops,
- GPIO signal timing.

156

158

It requires no initialization and is automatically incremented by hardware.

160

#### Base Address (for Pi Zero 2 W):

```
118  System Timer Base: 0x3F003000
    CLO (Low 32-bit Counter): 0x3F003004
120  CHI (High 32-bit Counter): 0x3F003008
```

#### Access (C Example):

```
122  #define SYS_TIMER_CLO ((volatile unsigned int*)(0x3F003004))
124
126  void delay_us(unsigned int us) {
    unsigned int start = *SYS_TIMER_CLO;
    while ((*SYS_TIMER_CLO - start) < us);
128  }
```

#### 4.5.2 ARM Generic Timer

The ARM Generic Timer is a high-precision 64-bit timer built directly into the ARMv8-A CPU core. It provides access to the current cycle count and its frequency via special system registers. Unlike peripheral-based timers (e.g., MMIO System Timer), the ARM Generic Timer operates independently of memory-mapped I/O and provides nanosecond-level resolution, making it ideal for time-critical applications.

162

164

166

168

#### Registers used:

- cntvct\_el0: Counter register –current value of the virtual timer.
- cntfrq\_el0: Frequency register –holds the frequency (in Hz) at which cntvct\_el0 increments.

170

172

The frequency is typically a fixed hardware value (e.g., 19.2 MHz on the Raspberry Pi). This means the counter increases by approximately 19.2 million ticks per second.

174

176

This provides reliable timing with 1 μs resolution.

### 4.4 cntvct\_el0

ARMv8 systems include a **Generic Timer** that can be read using the special register cntvct\_el0, which increments at the frequency specified by cntfrq\_el0.

#### Example (in C with inline ASM):

```
136  unsigned int get_cntfrq() {
    unsigned int freq;
138  asm volatile("mrs %0, cntfrq_el0" : "=r"(freq));
    return freq;
}

140  unsigned long long get_cntvct() {
    unsigned long long val;
    asm volatile("mrs %0, cntvct_el0" : "=r"(val));
    return val;
}
```

#### Reading the timer in C (bare-metal):

```
178  [language=C]
140  static inline uint64_t read_cntvct() {
    uint64_t value;
142  asm volatile ("mrs %0, cntvct_el0" : "=r"(value));
    return value;
}
```

**Use cases:**

- GPIO pulse-width generation.
- Performance measurement (cycle counters).
- OS scheduling (bare-metal or hypervisor environments).

**Summary:** The ARM Generic Timer is the most accurate and efficient timing mechanism available in ARMv8-based systems like the Raspberry Pi Zero 2 W. Its direct register access, high frequency, and isolation from MMIO make it essential for precise timekeeping in bare-metal applications.

**Sources****References**

- [1] Broadcom, *Bcm2835 arm peripherals*, Revision 6, <https://datasheets.raspberrypi.com/bcm2835/bcm2835-peripherals.pdf>, 2012.

**Creating a microsecond delay:**

```
[language=C]
void delay_us(uint64_t microseconds) {
    uint64_t freq = read_cntfrq(); // e.g., 19200000 Hz
    uint64_t ticks = (freq / 1000000) * microseconds;
    uint64_t start = read_cntvct();
    while ((read_cntvct() - start) < ticks);
}
```

**Advantages of ARM Generic Timer:**

- **High Resolution:** Nanosecond precision is possible depending on clock.
- **Reliable and Consistent:** Unlike MMIO-based timers, no memory-mapped I/O overhead.
- **Independent of peripherals:** Works solely in CPU core space –immune to peripheral resets or changes.
- **Ideal for profiling:** Allows benchmarking and profiling of short function calls or routines.

**Practical Considerations:**

- Ensure the timer is enabled by default (usually done by firmware). In most Raspberry Pi SoCs, this is already the case.
- The counter is 64-bit and free-running; no wraparound for hundreds of years at typical frequencies.
- If running with EL1/EL2/EL3, access may need to be enabled using system control registers like CNTKCTL\_EL1.

**Sample Assembly Delay (approximate):**

```
// Delay ~1 millisecond assuming 19.2 MHz
mrs x0, cntfrq_el0          // Get frequency
lsr x0, x0, #10             // Divide by ~1000 (approx)
mrs x1, cntvct_el0          // Get current timer
add x0, x0, x1              // Target tick
wait_loop:
mrs x2, cntvct_el0
cmp x2, x0
b.lo wait_loop
```