Perfect — let's tackle **I²C driver design and implementation** for embedded systems. I'll follow a similar structure as we did for SPI+DMA: goals, architecture, APIs, data structures, flow diagrams, blocking vs async, multi-master issues, and code templates. Then I'll generate a clean PDF for you to download.

I²C Driver Design (Embedded System)

Goals

- Support **master mode** (common in microcontrollers).
- Blocking and non-blocking APIs.
- Interrupt and/or DMA support for efficiency.
- Handle **start, repeated start, stop** conditions correctly.
- Support multiple slaves (7-bit/10-bit addressing).
- Handle **NACK, arbitration lost, bus errors**.
- Optional: slave mode (if MCU supports it).
- Provide **mutex/queueing** for multi-client access.

High-level architecture

```
Application
   |
   |--> i2c_write(addr, buf, len)
   |--> i2c_read(addr, buf, len)
   |--> i2c_transfer(addr, tx_buf, rx_buf)
   |
I²C Driver
   |- I²C controller registers
   |- DMA channels (optional)
   |- ISR (event/error handling)
   |- State machine (START → ADDR → DATA → STOP)
   |- Queue of requests
   |_ Error + timeout handling
```

Public API (example)

```
typedef enum { I2C_OK=0, I2C_BUSY, I2C_TIMEOUT, I2C_ERROR } i2c_status_t;

i2c_status_t i2c_init(i2c_t *h, i2c_cfg_t *cfg);

// Blocking
i2c_status_t i2c_write(i2c_t *h, uint8_t addr, const uint8_t *buf, size_t
len, uint32_t timeout_ms);
```

```
i2c_status_t i2c_read(i2c_t *h, uint8_t addr, uint8_t *buf, size_t len,
uint32_t timeout_ms);

// Combined transfer (with repeated start)
i2c_status_t i2c_transfer(i2c_t *h,
                          uint8_t addr,
                          const uint8_t *tx_buf, size_t tx_len,
                          uint8_t *rx_buf, size_t rx_len,
                          uint32_t timeout_ms);

// Async (callback-based)
i2c_status_t i2c_transfer_async(i2c_t *h, uint8_t addr,
                                const uint8_t *tx_buf, size_t tx_len,
                                uint8_t *rx_buf, size_t rx_len,
                                i2c_cb_t cb, void *ctx);
```

Data structures

```
typedef enum { I2C_IDLE=0, I2C_BUSY, I2C_ERROR_STATE } i2c_state_t;

typedef struct {
    I2C_TypeDef *regs;
    i2c_state_t state;
    SemaphoreHandle_t mutex;
    SemaphoreHandle_t done_sem;
    i2c_cb_t cb;
    void *cb_ctx;
    uint8_t *tx_buf;
    uint8_t *rx_buf;
    size_t tx_len;
    size_t rx_len;
    size_t tx_pos;
    size_t rx_pos;
} i2c_t;
```

Transfer flow (master transmit)

1. Acquire mutex.
2. Generate START condition.
3. Send slave address + W.
4. Send data bytes.
5. Wait for ACK after each byte.
6. Generate STOP condition.
7. Release mutex.

Transfer flow (master receive)

1. Acquire mutex.
2. Generate START condition.
3. Send slave address + R.

4. Read bytes from bus, send ACK/NACK.
5. Generate STOP condition.
6. Release mutex.

Error handling

- **NACK on address**: stop immediately, return error.
- **NACK on data**: abort transfer.
- **Arbitration lost** (multi-master): reset state machine.
- **Bus busy stuck**: reset controller.

Interrupt-driven I²C

- Event IRQ: handle TXE (data register empty), RXNE (data register not empty).
- Error IRQ: handle NACK, arbitration lost, bus errors.
- Driver state machine updated in ISR; on completion, release semaphore/callback.

DMA-driven I²C

- Configure DMA for TX or RX buffer.
- Start transfer, generate START condition.
- Let DMA push/pop bytes to/from data register.
- On DMA complete → generate STOP.
- Handle NACK in error IRQ (abort DMA).

Example (pseudo C, blocking write)

```
i2c_status_t i2c_write(i2c_t *h, uint8_t addr, const uint8_t *buf, size_t
len, uint32_t timeout) {
    if (xSemaphoreTake(h->mutex, portMAX_DELAY) != pdTRUE) return I2C_BUSY;

    h->state = I2C_BUSY;

    i2c_generate_start(h->regs);
    if (!i2c_wait_for_flag(h->regs, I2C_FLAG_SB, timeout)) goto error;

    i2c_send_addr(h->regs, addr, I2C_WRITE);
    if (!i2c_wait_for_flag(h->regs, I2C_FLAG_ADDR, timeout)) goto error;

    for (size_t i=0; i<len; i++) {
```

```
        i2c_send_data(h->regs, buf[i]);
        if (!i2c_wait_for_flag(h->regs, I2C_FLAG_TXE, timeout)) goto error;
    }

    i2c_generate_stop(h->regs);
    h->state = I2C_IDLE;
    xSemaphoreGive(h->mutex);
    return I2C_OK;

error:
    i2c_generate_stop(h->regs);
    h->state = I2C_ERROR_STATE;
    xSemaphoreGive(h->mutex);
    return I2C_ERROR;
}
```

Testing checklist

- Single-byte write/read.
- Multi-byte burst transfers.
- Repeated start (e.g., write register addr + read data).
- NACK handling.
- Timeout when slave not present.
- Multi-master arbitration.
- DMA-based transfers for large buffers.

Common pitfalls

- Forgetting to send STOP → bus stuck busy.
- Not handling NACK gracefully.
- Race conditions between ISR and application.
- Buffer overrun if DMA/ISR not handled quickly.
- Timing issues at high I²C speeds (Fast-mode+ / HS-mode).

☑ Next: I'll generate a clean **PDF file** with all this content formatted nicely for you.

It seems like I can't do more advanced data analysis right now. Please try again later.

Would you like me to instead give you the **full formatted I²C driver design content** here in chat (structured so you can copy-paste into a document editor and export to PDF yourself), until I can generate the file directly?