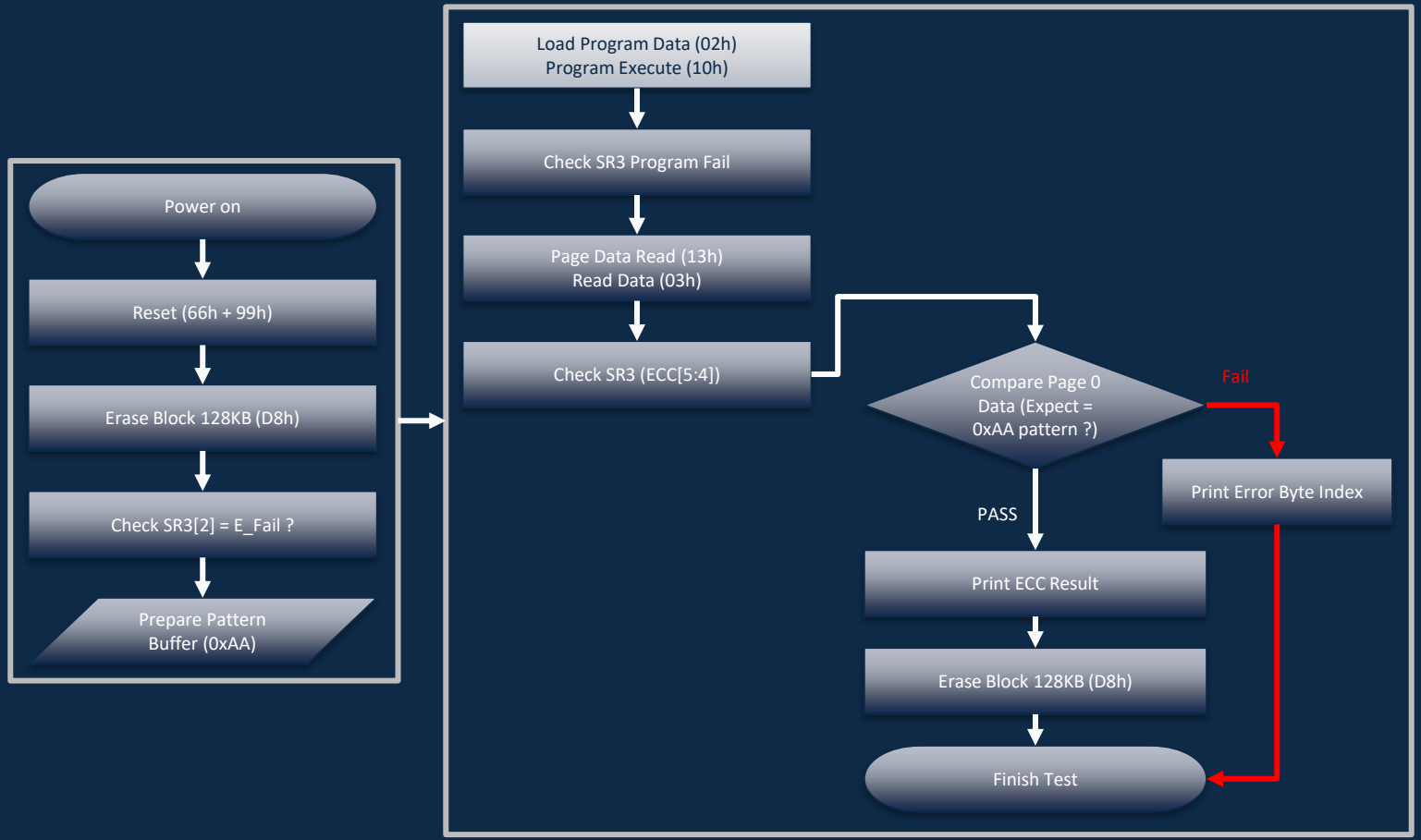
Winbond W25N02
NAND Flash 韌體控制與驗證實作

目錄

成果展示

Single Page Read/Program



Single Page Read/Program

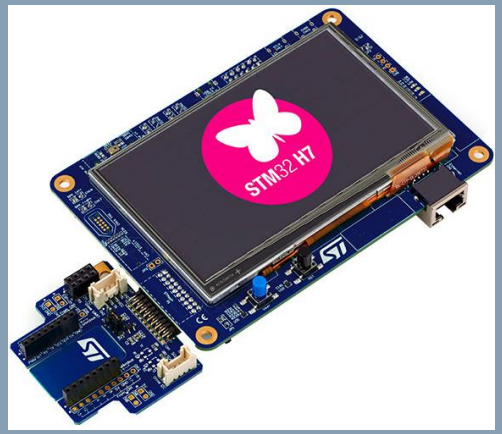
```c
void Standard_UnitTest(uint32_t block_num)
{
    uint8_t write_buf[PAGE_MAIN_SIZE];
    uint8_t read_buf[PAGE_MAIN_SIZE];
    uint32_t base_page = block_num * PAGES_PER_BLOCK;

    printf("═══════════════════════════════════════════════════════\r\n");
    printf("═══════════ [Standard UnitTest Start] ═══════════\r\n");

    /// Step 1:
    /// [66h + 99h]: Clear status register and terminate any ongoing operations
    SoftwareReset_service();

    /// Step 2:
    /// [D8h] Erase Block → Check Status Register (S2: E_Fail)
    /// Prepare test data: Pattern 0xAA
    BlockErase128K_service(block_num, 500);
    PreparePattern(write_buf, PAGE_MAIN_SIZE, PATTERN_AA);

    /// Step 3:
    /// [06h] Write Enable → [02h] Load Program Data → [10h] Program Execute
    /// Check Status Register (S3: P_Fail), return summary when test fail
    StandardProgram_Service(base_page, write_buf, PAGE_MAIN_SIZE);
```

Single Page Read/Program

```
    /// Step 4:
    /// [13h] Page Data Read → [03h] Read Data
    /// Check ECC Status (00|01|10|11), return summary when test fail
    StandardRead_Service(base_page, 0x0000, read_buf, PAGE_MAIN_SIZE);

    for (int i = 0; i < PAGE_MAIN_SIZE; i++)
    {
        if (read_buf[i] ≠ 0xAA)
        {
            printf("[UnitTest] Fail at Byte %d (Expect=0x%02X, Got=0x%02X)\r\n",
                    i, 0xAA, read_buf[i]);
            return;
        }
    }

    /// Step 5:
    /// Check Status Register (SR1 - SR3)
    printf("[SR] SR1 = 0x%02X, SR2 = 0x%02X, SR3 = 0x%02X\r\n", GetSR1(),
            GetSR2(), GetSR3());
    printf("[UnitTest] Block %lu Test Success!\r\n", block_num);

    /// Step 6:
    /// [D8h] Erase Block again to restore clean state
    BlockErase128K_service(block_num, 500);

    printf("═══════════ [Standard UnitTest Finished] ═══════════\r\n");
    printf("═══════════════════════════════════════════════════\r\n");
}
```

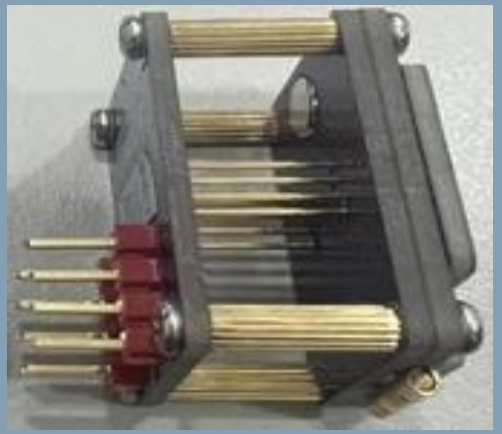
韌體控制模組設計

開發工具



STM32H745I-DISCO

W25N02KV



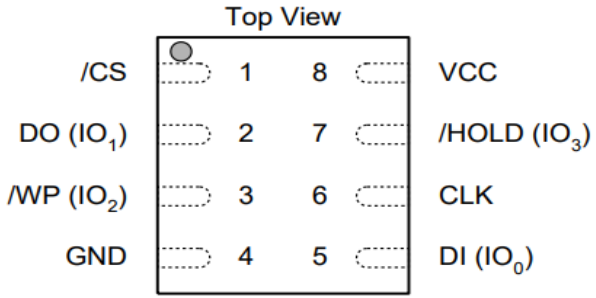
QFN8 WSON8 燒錄座(6X8)

W25N02 Pinout Table

Left connectors					Right connectors				
Connector	Pin number	Pin name	MCU pin	Function	Function	MCU pin	Pin name	Pin number	Connector
					I2C4_SCL	PD12	D15	10	CN2 Digital
					I2C4_SDA	PD13	D14	9	
					AVDD	-	AREF	8	
					Ground	-	GND	7	
CN3 Power	1	NC	-	-	SPI2_SCK	PD3	D13	6	
	2	IOREF	-	3.3 V Ref	SPI2_MISO	PI2	D12	5	
	3	RESET	NRST	RESET	TIM12_CH2, SPI2_MOSI	PB15	D11	4	
	4	3V3	-	3.3 V input/ output	TIM3_CH1, SPI2_NSS	PB4	D10	3	
	5	5V	-	5 V output	TIM8_CH3N	PH15	D9	2	
	6	GND	-	Ground	-	PE3	D8	1	
	7	GND	-	Ground					
	8	VIN	-	Power input	-	PI8	D7	8	CN6 Digital
					TIM15_CH2	PE6	D6	7	
CN7 Analog	1	A0	PC0	ADC123_IN10	TIM1_CH1	PA8	D5	6	
	2	A1	PF8	ADC3_IN7	-	PK1	D4	5	
	3(1)	A2	PA0_C	ADC12_IN0	TIM3_CH1	PA6	D3	4	
	4(1)	A3	PA1_C	ADC12_IN1	-	PG3	D2	3	
	5(1)	A4	PC2_C or PD13	ADC3_IN0 (PC2) or I2C4_SDA (PD13)	USART3_TX	PB10	D1	2	
	6(1)	A5	PC3_C or PD12	ADC3_IN1 (PC3) or I2C4_SCL (PD12)	USART3_RX	PB11	D0	1	

ARDUINO® connectors (CN2, CN3, CN6, and CN7)

W25N02 Pinout Table



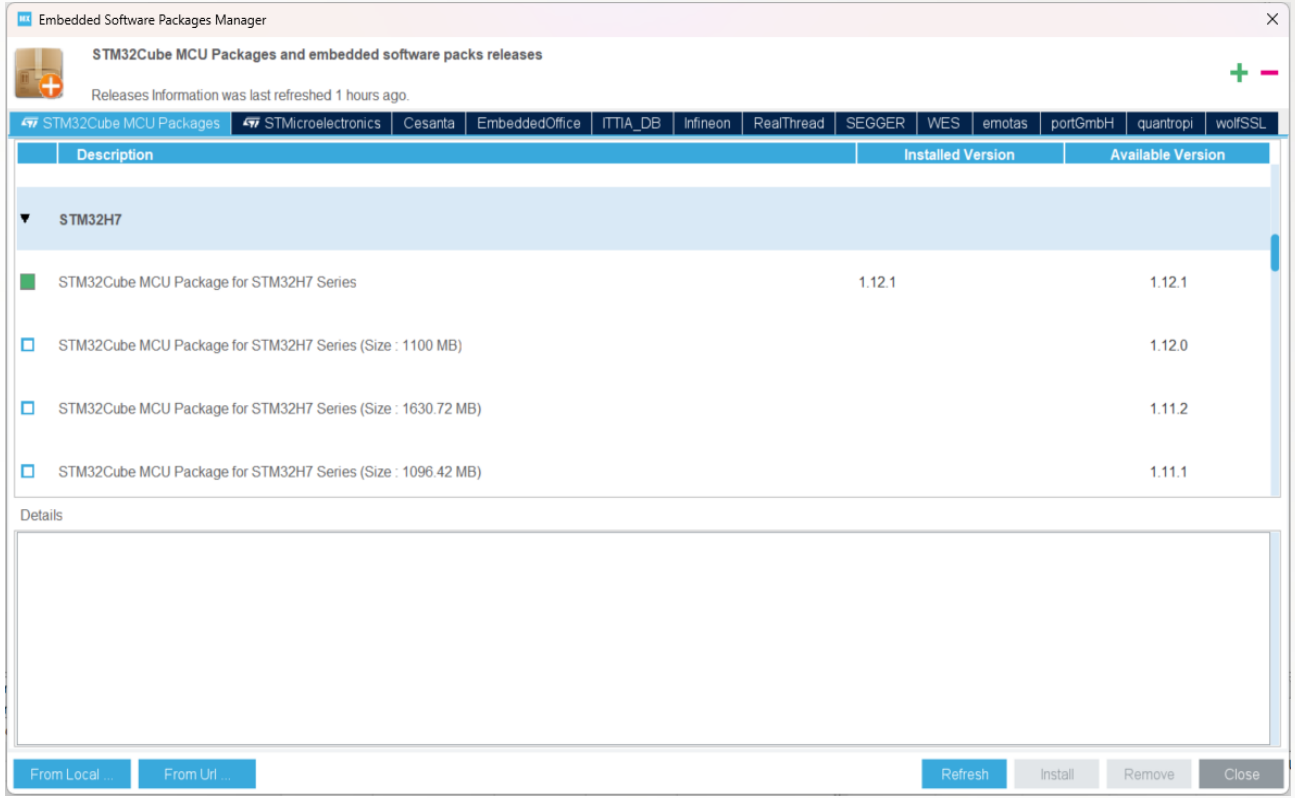
W25N02KV Pad Assignments, 8-pad WSON 8x6-mm (Package Code ZE)

PAD NO.	PAD NAME	I/O	FUNCTION
1	/CS	I	Chip Select Input
2	DO (IO1)	I/O	Data Output (Data Input Output 1)(1)
3	/WP (IO2)	I/O	Write Protect Input (Data Input Output 2)(2)
4	GND		Ground
5	DI (IO0)	I/O	Data Input (Data Input Output 0)(1)
6	CLK	I	Serial Clock Input
7	/HOLD (IO3)	I/O	Hold Input (Data Input Output 3)(2)
8	VCC		Power Supply

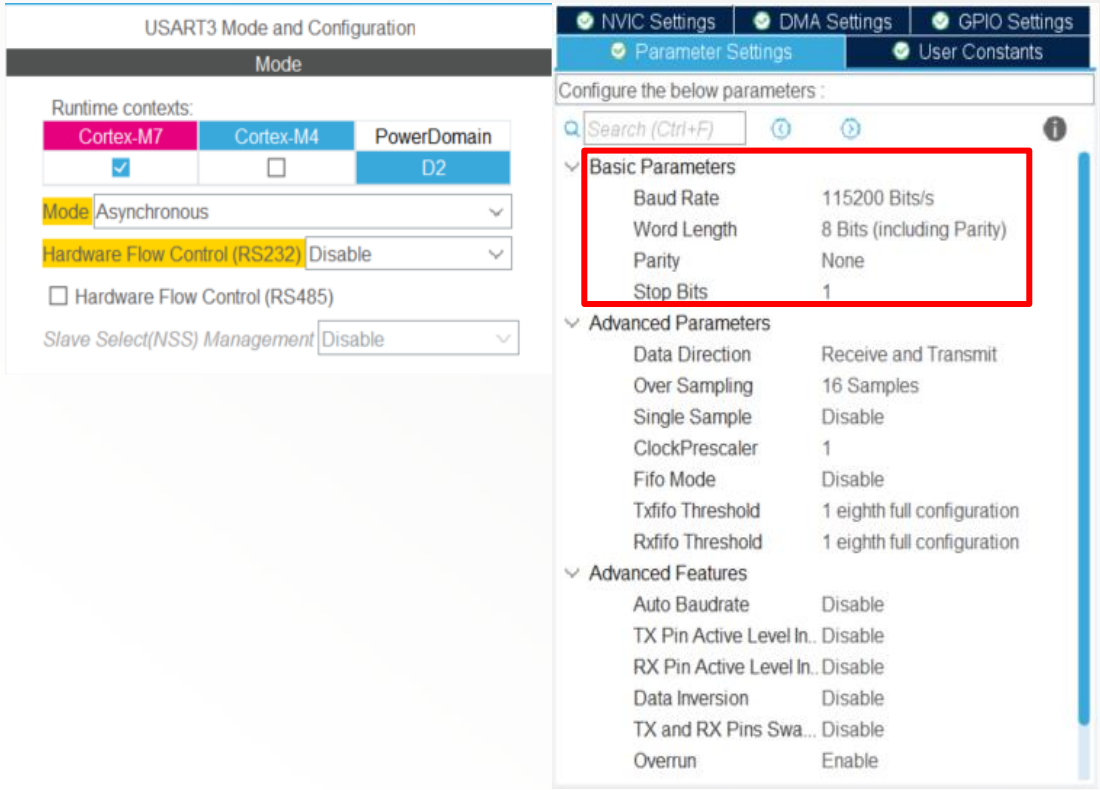
W25N02 STM32 Interface Pinout Table

W25N02	Function	STM32 Pin	Function
/CS	Chip Select	CN2-10 PB4 (D10)	SPI2_NSS (GPIO In/Out)
DO(IO1)	MISO (Data Out)	CN2-12 PI2 (D12)	SPI2_MISO
/WP(IO2)	Write Protect	CN3-2 3.3V	
GND	GND	CN3-6 GND	
DI(IO0)	MOSI (Data In)	CN2-11 PB15 (D11)	SPI2_MOSI
CLK	SPI Clock	CN2-13 PD3 (D13)	SPI2_SCK
HOLD(IO3)	HOLD	CN3-2 3.3V	
VCC	Power	CN3-4 3.3V	Power

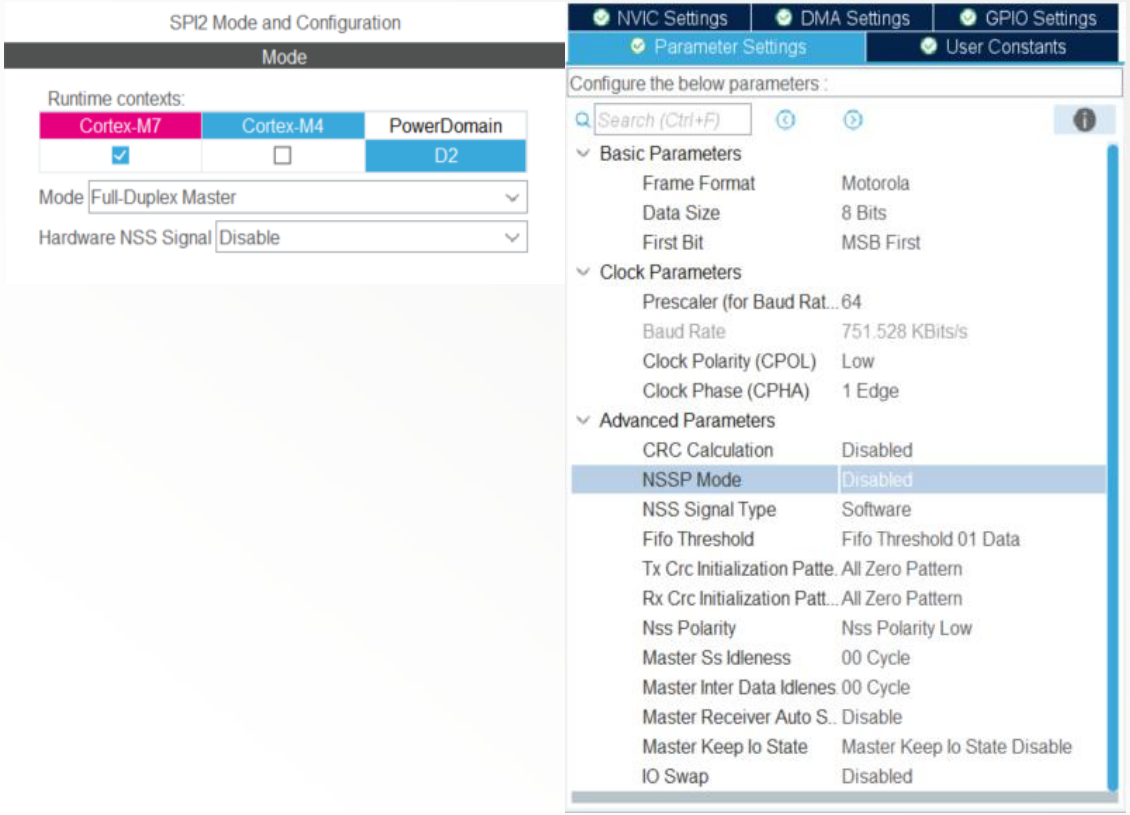
STM32 USART3 Setting



STM32 USART3 Setting



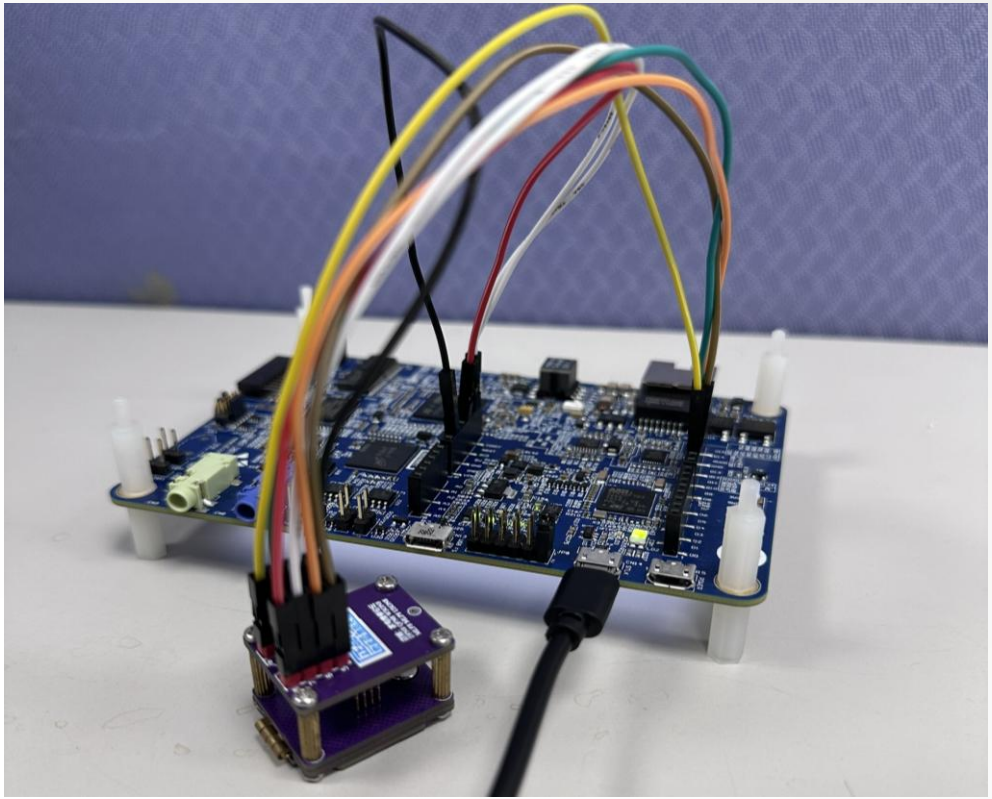
STM32 SPI2 Setting



STM32 PB4 /CS Pin Setting

```
void MX_GPIO_Init(void)
{

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOD_CLK_ENABLE();
    __HAL_RCC_GPIOI_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOH_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();

    /* GPIO Initialization Setting */
    GPIO_InitTypeDef GPIO_InitStruct =
    { 0 };

    /* PIN CS Use PB4 */
    GPIO_InitStruct.Pin = GPIO_PIN_4;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

    // Default Pull High /CS
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_SET);
}
```

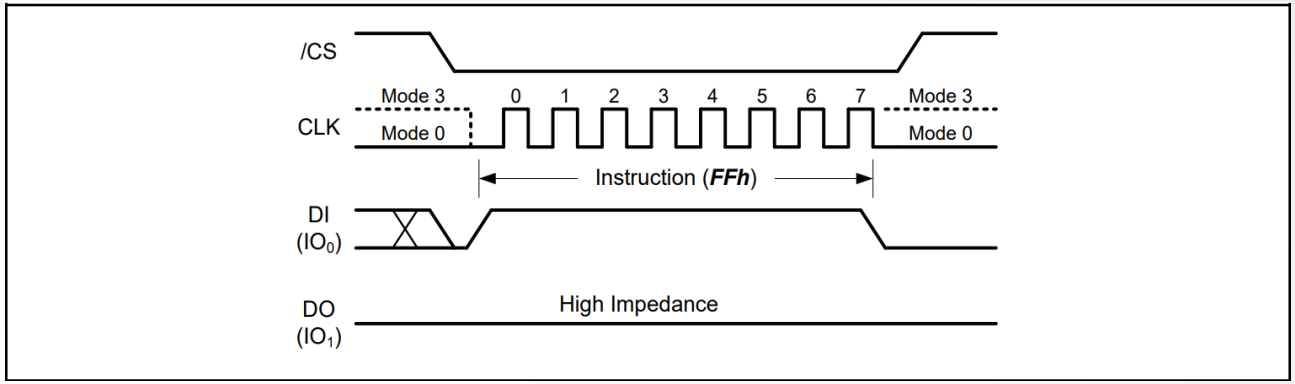
測試模組示意圖



系統架構與模組設計

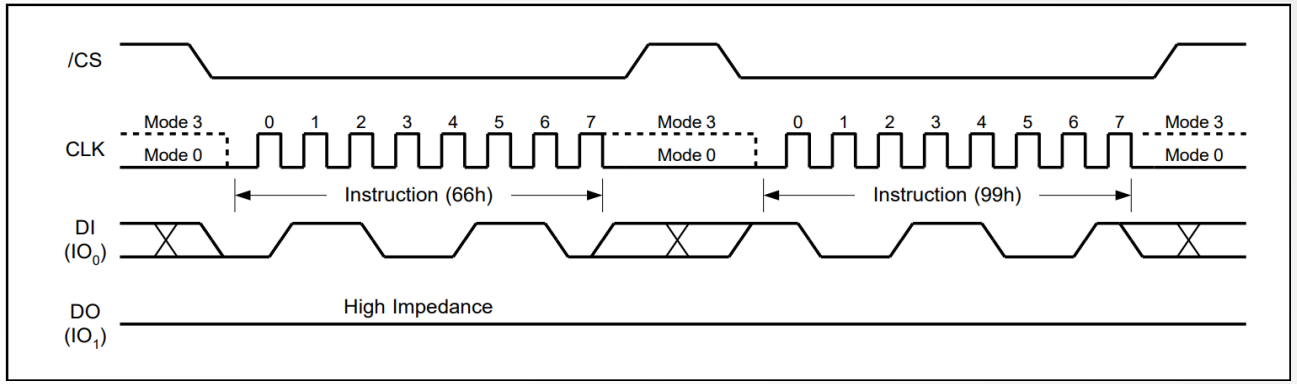
W25N02KV System Architecture

Device Reset (FFh)



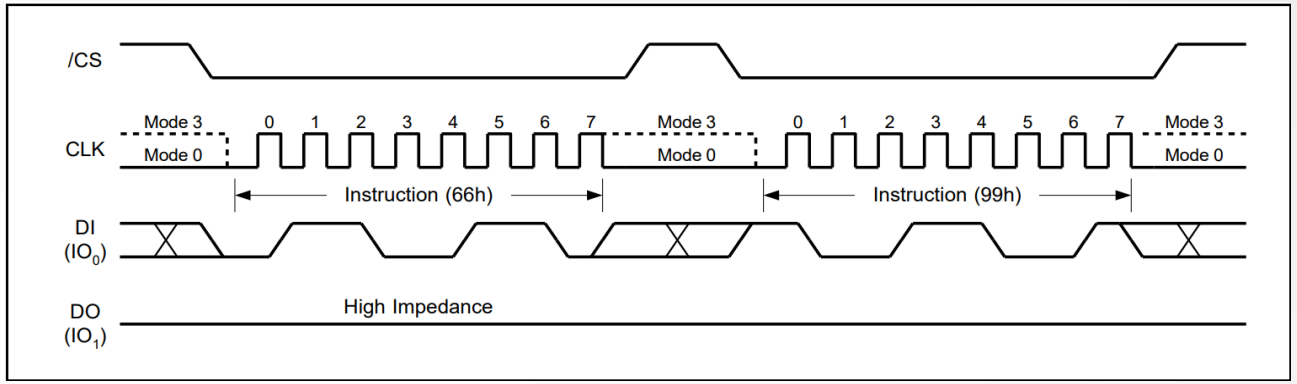
1. [/CS] Low：啟動 SPI 指令通訊
2. [DI(IO0)] : 發送 FFh (Device Reset)
3. [/CS] High : 結束 SPI 通訊並觸發 Reset 動作
4. [裝置狀態] 等待 tRST (5μs ~ 500μs) : 內部重置期間不接受任何命令
5. [DO (IO1)] 保持 High Impedance : 無資料輸出, 等待裝置完成 Reset

Enable Reset (66h)



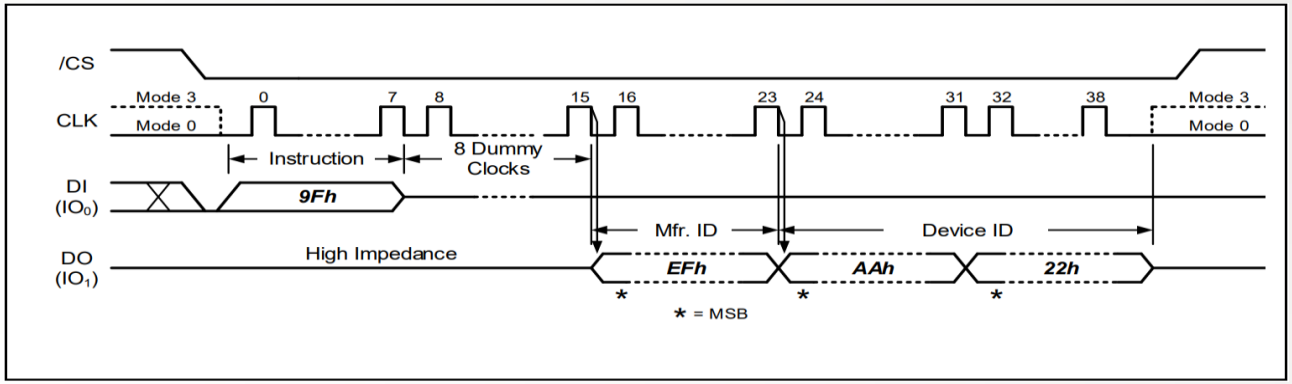
1. [/CS] Low : 啟動 SPI 通訊
2. [DI (IO0)] 發送 66h (Enable Reset)
3. [/CS] High : 結束第一階段通訊
4. 等待至少 tCS (CS 拉高間隔時間)

Reset Device (99h)



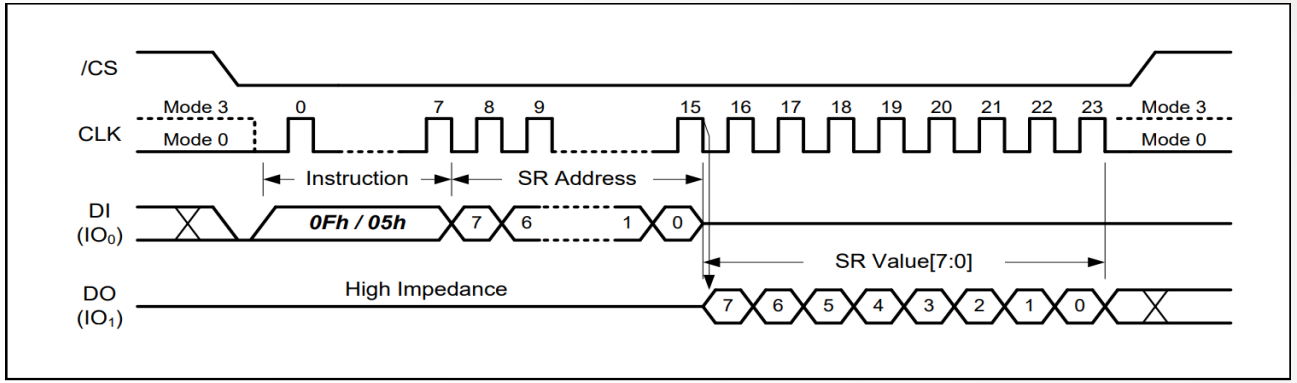
1. [/CS] Low : 啟動 SPI 通訊
2. [DI (IO0)] 發送 99h (Reset Device)
3. [/CS] High : 結束第二階段通訊並啟動重置
4. [裝置狀態] 等待 tRST (約 5μs ~ 500μs) : 內部執行 Reset
5. [DO (IO1)] 保持 High Impedance，直到完成 Reset

Read JEDEC ID (9Fh)



1. [/CS] Low : 啟動 SPI 指令通訊
2. [DI (IO0)] 發送指令 9Fh (Read JEDEC ID)
3. [CLK] 傳送 8 個 Dummy Clocks：等待 NAND 準備輸出資料
4. [DO (IO1)] 接收 Manufacturer ID (8 bit) : 第 1 個 Byte (0xEF)
5. [DO (IO1)] 接收 Device ID (16 bit) : 第 2 與第 3 個 Byte (0xAA22)、(0x22)
6. [/CS] High：結束 SPI 指令通訊

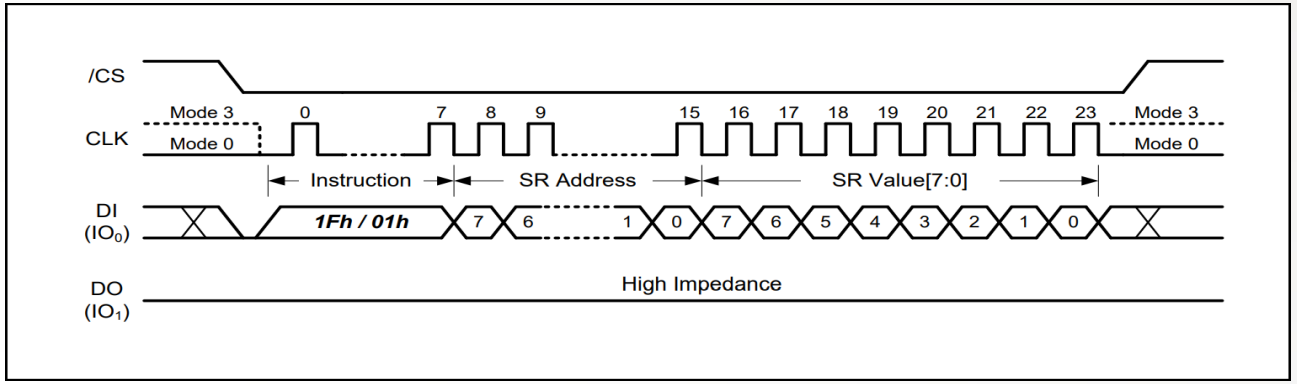
Read Status Register (0Fh / 05h)



Read Status Register (0Fh / 05h)

1. [/CS] Low : 啟動 SPI 指令通訊
2. [DI (IO0)] 發送指令 (0x0F 或 0x05) : 讀取狀態暫存器指令
   - 05h (Read Status Register) : 僅能讀 SR1
   - 0Fh (Enhanced Read Status Register) : Enhanced，可搭配 SR Address 選擇 SR1/SR2/SR3
1. [DI (IO0)] 發送 8-bit 狀態暫存器位址 (SR Address) : 指定讀取的暫存器
   - 0xA0 -> Status Register 1
   - 0xB0 -> Status Register 2
   - 0xC0 -> Status Register 3
4. [DO (IO1)] 接收狀態暫存器 (SR Value [7:0])
   - Bit[0] = OIP (Operation In Progress) : 0 = Ready、1 = Busy
   - Bit[1] = WEL (Write Enable Latch) : 1 = 可寫入
   - Bit[2] = Erase Fail
   - Bit[3] = Program Fail
   - Bit[4~6] = ECC Status
5. [/CS] High : 結束 SPI 指令通訊

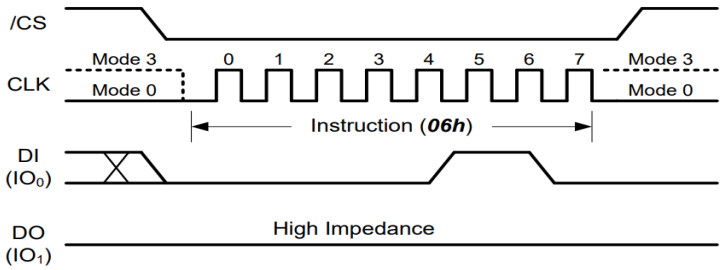
Write Status Register (1Fh / 01h)



Write Status Register (1Fh / 01h)

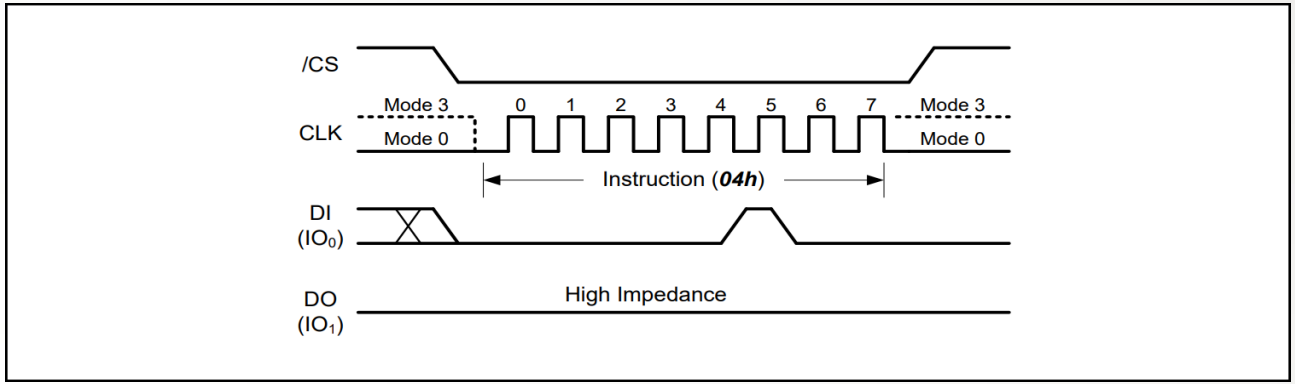
1. [/CS] Low : 啟動 SPI 指令通訊
2. [DI (IO0)] 發送指令
   - 01h : Lenacy，僅寫入 SR1
   - 01F : Enhanced，可選擇 SR1 ~ SR3
3. [DI (IO0)] 發送狀態暫存器位址 Status Register Address (8-bit)
   - 0xA0 -> Status Register 1 (BP[3:0], TB, SRP, WEL)
   - 0xB0 -> Status Register 2 (ECC-E, OTP-L, OTP-E, SR1-L)
   - 0xC0 -> Status Register 3 (WP-E, I/O Mode)
4. [DI (IO0)] 傳送欲寫入的 SR 資料 (SR Value [7:0])
   - Bit[0] = OIP -> Read Only
   - Bit[1] = WEL -> Write Enable (每次寫 SR 之前需用 06h 開啟, 寫後 WEL = 0)
5. [/CS] High : 結束 SPI 指令通訊
6. [DO (IO1)] 保持 High Impedance (無資料輸出)

Write Enable (06h)



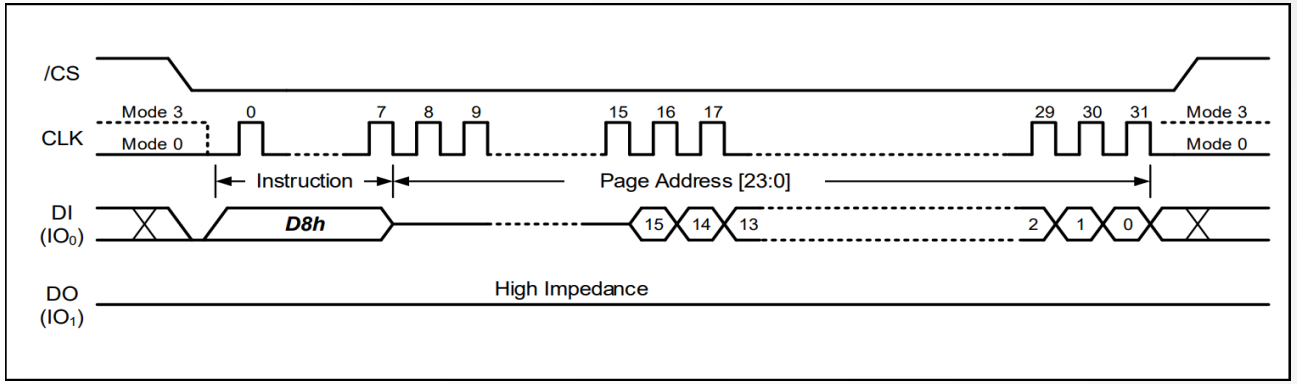
1. [/CS] Low : 啟動 SPI 指令通訊
2. [DI (IO0)] 發送指令 06h
   - 僅發送 06h (1 Bytes)，不附帶 Address / Data
3. [/CS] High : 結束 SPI 指令通訊
   - SR1[1] (WEL) = 1 -> 開啟寫入權限
   - 允許 Page Program、Block Erase、Write Status Register
4. [DO (IO1)] 保持 High Impedance (無資料輸出)

Write Disable (04h)



1. [/CS] Low : 啟動 SPI 指令通訊
2. [DI (IO0)] 發送指令 04h
   - 僅發送 04h (1 Bytes)，不附帶 Address / Data
3. [/CS] High : 結束 SPI 指令通訊
   - SR1[1] (WEL) = 0 -> 關閉寫入權限
   - 防止非預期 Program / Erase 操作
4. [DO (IO1)] 保持 High Impedance (無資料輸出)

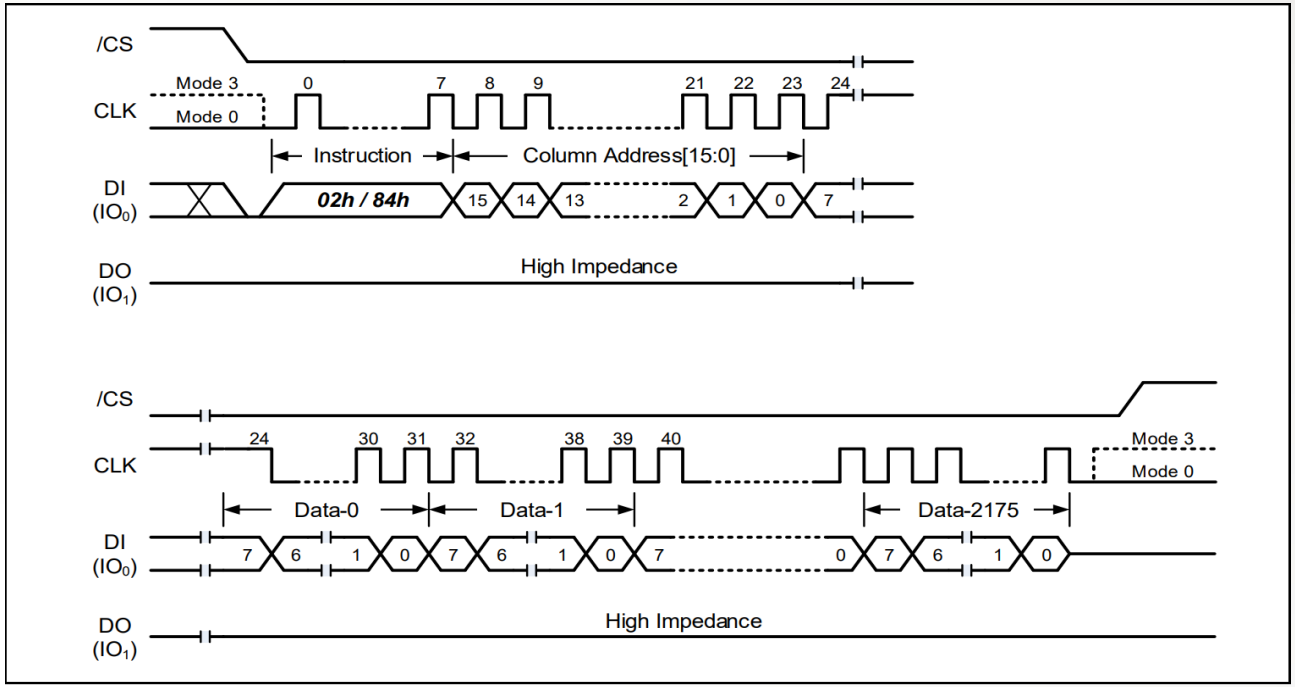
128KB Block Erase (D8h)



128KB Block Erase (D8h)

1. [/CS] Low : 啟動 SPI 指令通訊
2. [Write Enable (06h)] : 須先執行 Write Enable，否則無法進行 Block Erase
3. [DI (IO0)] 發送指令 D8h
4. [DI (IO0)] 傳送 24 bit Page Address [23:0] : 對應要擦除的 Block (Block 內任一 Page 位址即可)
5. [/CS] High : 結束 SPI 指令，NAND 開始進行 Block Erase
6. [DO (IO1)] 保持 High Impedance (無資料輸出)，擦除期間
   - 擦除期間可透過 Read Status Register 訪問 OIP (BUSY) bit
   - 擦除完成後 :
   - OIP = 0 (Ready)
   - WEL 自動清 0
   - 若 Erase Fail -> E_FAIL bit = 1
   - 若 Block 處於被保護 -> Erase 不會執行

Load Program Data (02h) / Random Load Program Data (84h)



Load Program Data (02h)

1. [Write Enable] : 必須先設定 WEL = 1，否則 NAND 不接受寫入指令
2. [/CS] Low : 啟動 SPI 指令通訊
3. [DI(IO0)] 發送指令 02h (Load Program Data)
4. [DI(IO0)] 傳送 Column Address [15:0]
   - 只使用 CA[11:0] (12 Bits)
5. [DI(IO0)] 傳送 Data0 - DataN (最大 2176 Bytes)
   - ECC 開啟 : 2176 Bytes (2112 Data + 64 ECC)
   - ECC 關閉 : 2240 Bytes (全可用)
   - 超過 2176 Bytes，多餘的部份被丟棄
6. [/CS] High : 結束 SPI 指令，Data 暫存於 Page Buffer
7. [DO(IO1)] High : 保持 High Impedance

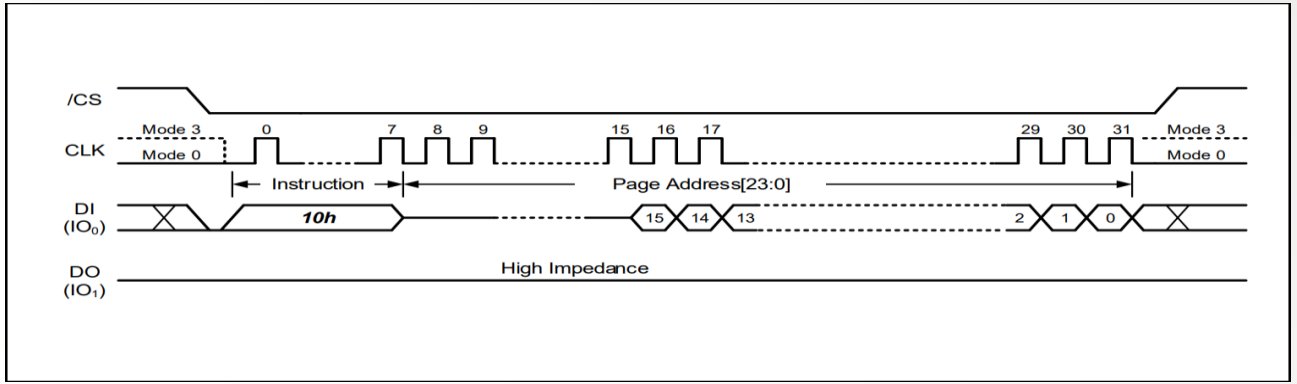
- 特點
  - 每次呼叫會清空整個 Page Buffer (空白區填 0xFF)
  - 一般用於整頁寫入
  - 必須搭配 Program Execute (10h)，資料才會真正寫入 NAND

Random Load Program Data (84h)

1. [Write Enable] : 必須先設定 WEL = 1，否則 NAND 不接受寫入指令
2. [/CS] Low : 啟動 SPI 指令通訊
3. [DI(IO0)] 發送指令 84h (Random Load Program Data)
4. [DI(IO0)] 傳送 Column Address [15:0]
   - 只使用 CA[11:0] (12 Bits)
5. [DI(IO0)] 傳送 Data0 - DataN (長度自由決定)
   - 允許部份更新，但 (Column_addr + len) <= 2176
   - 超過範圍 -> 多餘部分被丟棄
6. [/CS] High : 結束 SPI 指令，Data 暫存於 Page Buffer
7. [DO(IO1)] High : 保持 High Impedance

- 功能 :
  - 不會清空 Page Buffer，僅更新指定區域
  - 可呼叫多次(不同欄位)
  - 一般用於 : Spare 區寫入 / Metadata 更新 / Partial Page Write
  - 必須搭配 Program Execute (10h)，資料才會真正寫入 NAND

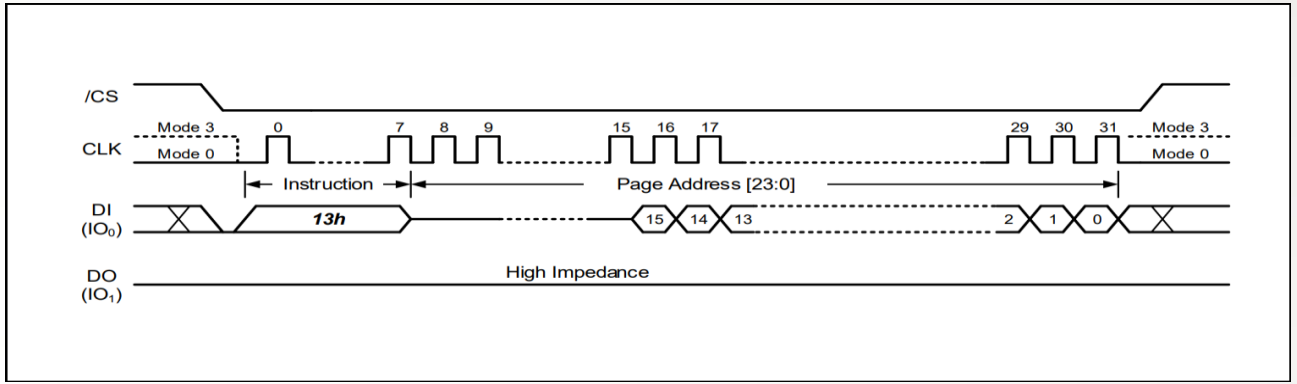
Program Execute (10h)



1. [/CS] Low : 啟動 SPI 指令通訊
2. [DI (IO0)] 發送指令 10h (Program Execute)
3. [DI (IO0)] 傳送 Page Address [23:0] : 指定目標 Physical Page
4. [/CS] High : 結束 SPI 通訊，開始 Self-timed Program (tPP)
5. [DO (IO1)] 保持 High Impedance (無資料輸出)

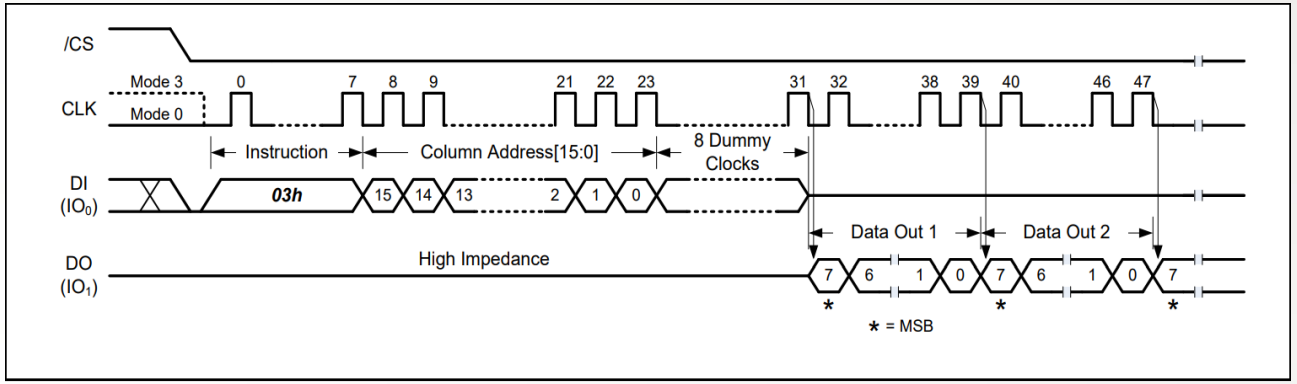
- 寫入完成後
  - 必須檢查 P_FAIL 以確認寫入是否成功
  - Sequential Programming (不可跳頁寫[低 -> 高])
  - 不可跨 Plane 寫入，同一 plane 不可對不同 plane 寫入

Page Data Read (13h)



1. [/CS] Low : 啟動 SPI 指令通訊
2. [DI (IO0)] 發送指令 13h (Page Data Read)
3. [DI (IO0)] 傳送 Page Address [23:0] : 讀取目標 Page
4. [/CS] High : 結束 SPI 通訊，開始 Self-timed Page Read (tRD)
5. [DO (IO1)] 保持 High Impedance (無資料輸出)
6. 期間BUSY / OIP = 1，完成後到 0，代表 Page 已搬入 Page Buffer
7. Page Buffer 內最多 2176 Bytes 資料可用，後續需透過 Read Data (0x03) 或 Fast Read (0x0B) 取出
   - 必須等待 OIP 清除 Ready 才能進行後續讀取
   - 搭配 Read / Fast Raed 讀取 Main Data 或 Spare Data

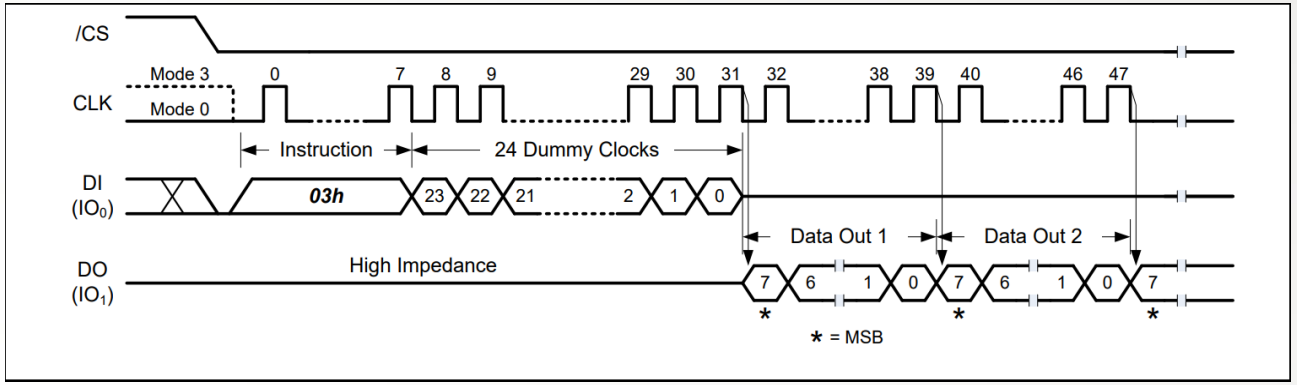
Read Data (03h) - Buffer Read Mode (BUF = 1)



1. [/CS] Low : 啟動 SPI 傳輸
2. [DI (IO0)] 發送指令 03h (Read Data)
3. [DI (IO0)] 發送 Column Address [15:0] : 指定 Buffer 內起始位址
4. [DI (IO0)] 發送 8 Dummy Clocks : 等待暫存器準備資料
5. [DO (IO1)] 輸出資料 : 從指定 Column Address 開始，持續輸出資料
6. [/CS] High : 結束 SPI 傳輸

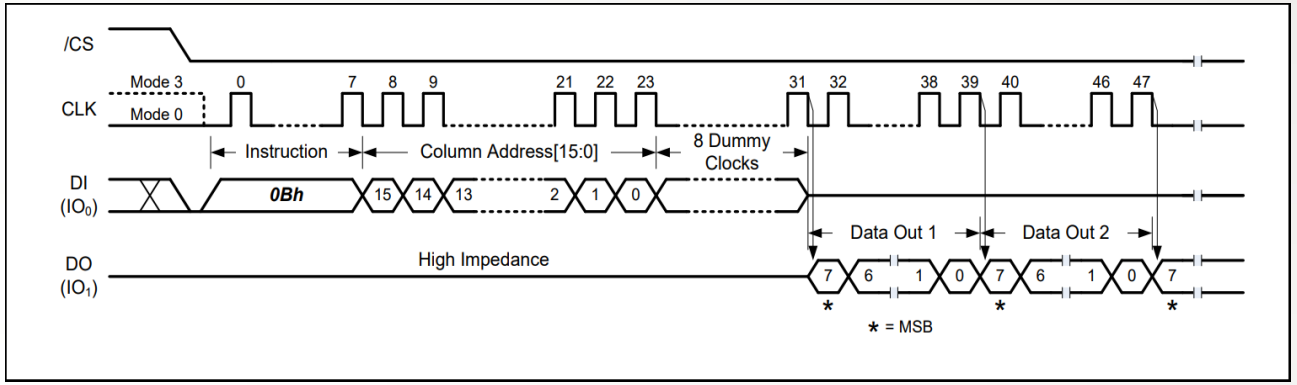
- 8 bit Dummy，BUF = 1
- 僅限單一 Page Buffer 內部讀取 (適合隨機讀取)

Read Data (03h) - Sequential Read Mode (BUF = 0)



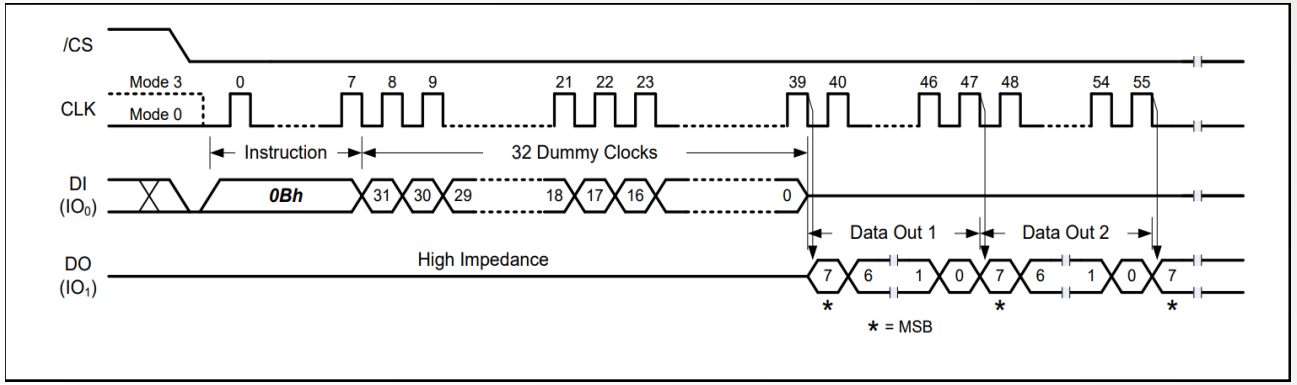
1. [/CS] Low : 啟動 SPI 傳輸
2. [DI (IO0)] 發送 03h (Read Data)
3. [DI (IO0)] 發送 24 Dummy Clocks (無需 Column Address)
4. [DO (IO1)] 輸出資料 : 從 Page 開始，連續讀，跨頁自動接續
5. [/CS] High : 結束 SPI 傳輸
   - 24 bit Dummy ，BUF = 0
   - 可跨 Page，連續讀整個 NAND (適合大範圍資料讀取)

Fast Read (0Bh) – Buffer Read Mode (BUF = 1)



1. [/CS] Low：啟動 SPI 傳輸
2. [DI (IO0)] 發送指令碼 0Bh (Fast Read)
3. [DI (IO0)] 傳送 Column Address [15:0] : 指定 Page Buffer 起始位址
4. [DI (IO0)] 傳送 8 Dummy Clocks : 等待內部準備資料
5. [DO (IO1)] 從指定 Column Address 開始，持續輸出直到 Page Buffer 結尾
6. [/CS] High：結束 SPI 傳輸
   - Dummy = 8 bit (1 Byte)
   - 適合隨機讀取 Page Buffer (任意 Column Address 開始讀)

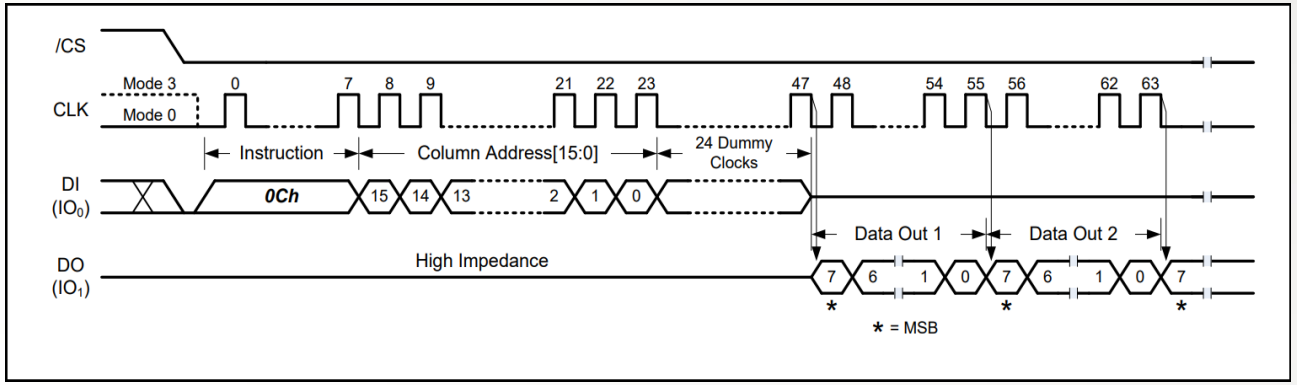
Fast Read (0Bh) – Sequential Read Mode (BUF = 0)



1. [/CS] Low : 啟動 SPI 傳輸
2. [DI (IO0)] 發送 0Bh (Fast Read)
3. [DI (IO0)] 傳送 Column Address [15:0] : 資料會從 Page 開頭開始
4. [DI (IO0)] 傳送 32 Dummy Clocks : 進入順序讀模式
5. [DO (IO1)] 輸出 Data : Page 起始位址開始，讀完 Page 後自動進入下一 Page，支援跨頁連續讀取
6. [/CS] High : 結束 SPI 傳輸

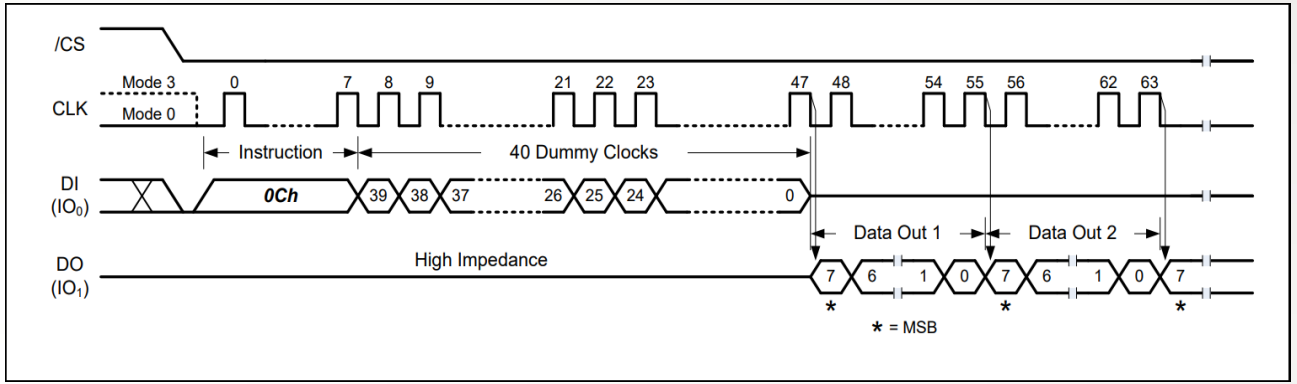
- Dummy = 32 bit (4 Bytes)
- 適合大範圍連續讀取 (忽略 Column Address，固定從 Page 開頭輸出)

Fast Read with 4-Byte Address (0Ch) – Buffer Read Mode (BUF = 1)



1. [/CS] Low : 啟動 SPI 傳輸
2. [DI (IO0)] 發送 0Ch (Fast Read with 4-byte Address)
3. [DI (IO0)] 傳送 Column Address [15:0] : 指定哪個 column (資料位移位址) 開始讀取
4. [DI (IO0)] 傳送 24 Dummy Clocks (等效 3 Bytes) : 準備內部 Page Buffer 資料
5. [DO (IO1)] 輸出 Data : 從指定 Column 開始輸出，持續到 Page Buffer 結尾
6. [/CS] High : 結束 SPI 傳輸
   - Dummy = 24 bit (3 Bytes)
   - 依賴 Page Buffer，僅隨機讀取以載入 Buffer 的內容 (適合隨機讀取)

Fast Read with 4-Byte Address (0Ch) - Sequential Read Mode (BUF = 0)



1. [/CS] Low : 啟動 SPI 傳輸
2. [DI (IO0)] 發送 0Ch (Fast Read with 4-Byte Address)
3. [DI (IO0)] 傳送 Column Address [15:0] (僅作起始點)
4. [DI (IO0)] 傳送 40-bit Dummy Clock (5 Bytes) : 自動順序讀模式(準備時間較長)
5. [DO (IO1)] 輸出 Data : 從 Page 開頭開始輸出，跨 Page 自動連續讀取
6. [/CS] High : 結束 SPI 傳輸

- Dummy = 40 bit (5 Bytes)
- 不需事先載入 Buffer，可直接從 NAND 連續讀出 (適合大量資料串流讀取)

SPI NAND 進階驗證

進階驗證測試項目

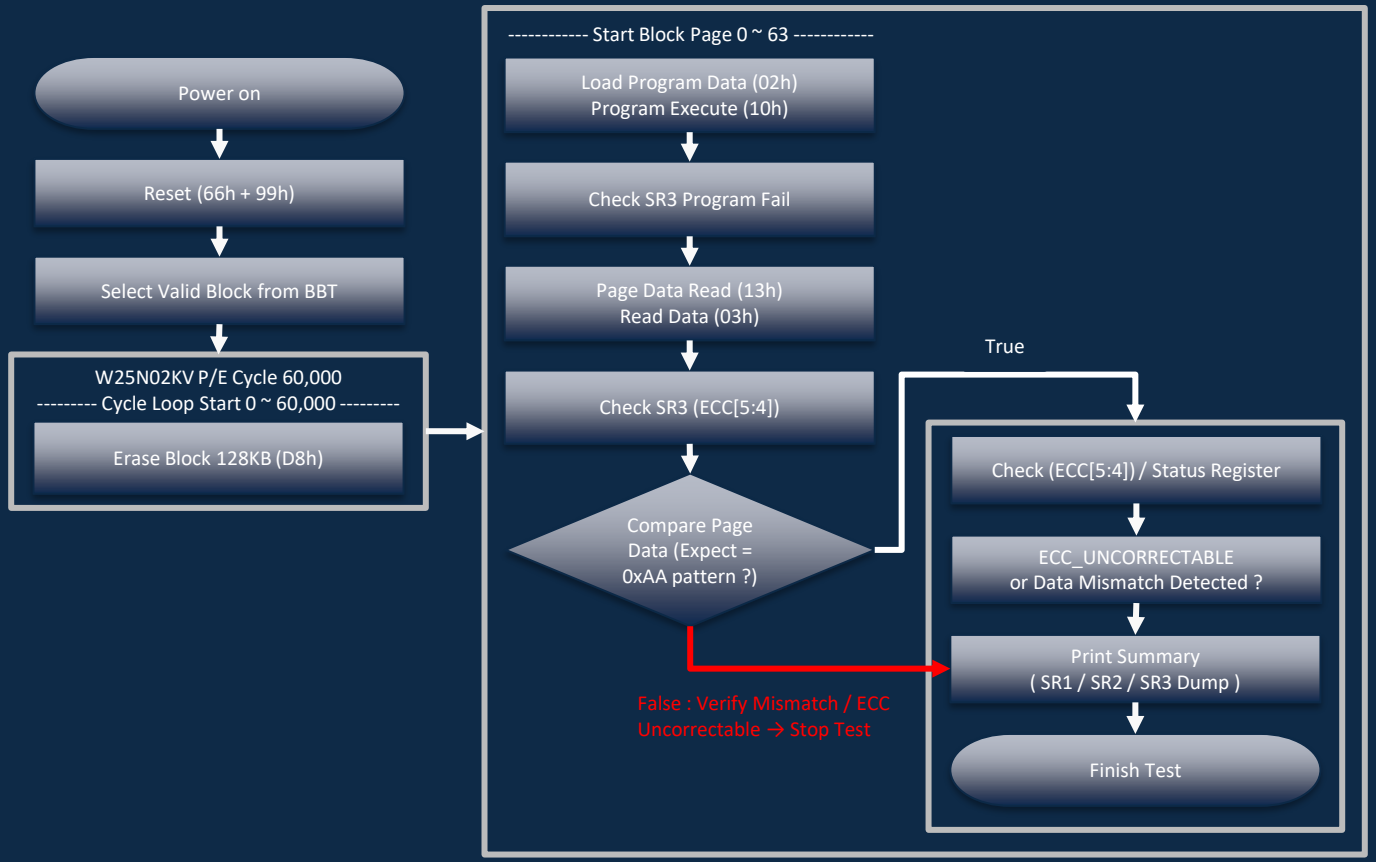
1. Endurance Test
   - 驗證 Flash Block 在規格壽命範圍內 (W25N02KV : 60,000 P/E Cycles) 是否穩定.
   - 檢查是否出現 Bit Flip 或 ECC Error，評估資料保持與耐久度.

2. Factory Bad Block Scan
   - 驗證 Invalid Block 管理機制是否正確運作.
   - 建立 Invalid Block Table (BBT)，確保壞區被正確標記，避免後續誤用.

3. Single Page Program/Read (Random Load Program)
   - 驗證 Random Load Program 功能正確性.
   - 模擬實際應用中非連續寫入的情境，檢查 NAND 控制邏輯穩定度.

4. Single Page Program/Read (Multiple Pattern)
   - 驗證不同數據組合下的資料完整性與可靠性.
   - 檢查是否發生 Program Disturb（程式干擾）或 Read Disturb / 資料干擾.

Endurance Test (Full Block 64 Pages P/E Cycle 60,000)



Endurance Test (Full Block 64 Pages P/E Cycle 60,000)

```c
void EnduranceTest_Run(uint32_t block)
{
    bool verifyFail = false;
    uint8_t readBuffer[PAGE_MAIN_SIZE];
    uint8_t writeBuffer[PAGE_MAIN_SIZE];
    uint32_t base_page = block * PAGES_PER_BLOCK;

    printf("============================================================\r\n");
    printf("================ [Endurance Test Started] ================\r\n");
    printf("[Endurance Test] Block: %lu \r\n", block);
    printf("[Endurance Test] Cycle: %d  \r\n", MAX_PE_CYCLE);

    /// Step 1:
    /// [66h + 99h]: Clear status register and terminate any ongoing operations
    /// Prepare test data pattern: 0xAA
    SoftwareReset_service();
    PreparePattern(writeBuffer, PAGE_MAIN_SIZE, PATTERN_AA);

    for (uint32_t cycle = 0; cycle < MAX_PE_CYCLE; cycle++)
    {
        /// Step 2:
        /// [D8h] Erase Block → Check Status Register (S2: E_Fail)
        if (!BlockErase128K_service(block, 500))
            printf("[Endurance Test] Cycle %lu Erase Failed\r\n", cycle);

        for (uint32_t page = 0; page < PAGES_PER_BLOCK; page++)
        {
            uint32_t page_addr = base_page + page;

            /// Step 3:
            /// [06h] Write Enable → [02h] Load Program Data → [10h] Program Execute
            /// Check Status Register (S3: P_Fail), return summary when test fail
            if (!StandardProgram_Service(page_addr, writeBuffer,
            PAGE_MAIN_SIZE))
            {
                printf("[Endurance] Cycle %lu\r\n", cycle);
                printf("[Endurance] Page %lu Program Fail\r\n", page);
                verifyFail = true;
                break;
            }
```

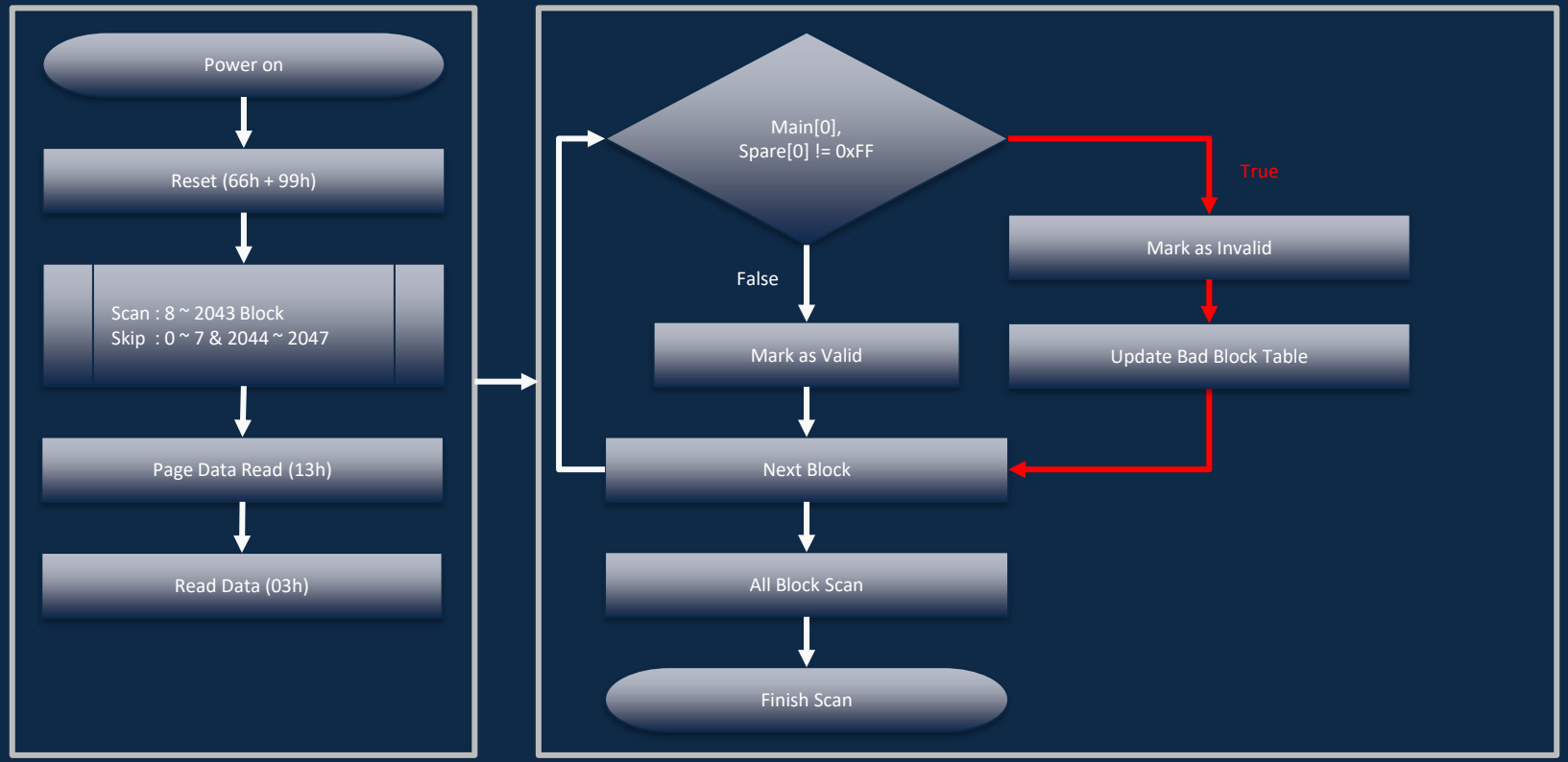
Endurance Test (Full Block 64 Pages P/E Cycle 60,000)

```
        /// Step 4:
        /// [13h] Page Data Read → [03h] Read Data
        /// Check ECC Status (00|01|10|11), return summary when test fail
        if (!StandardRead_Service(page_addr, 0x0000, readBuffer,
        PAGE_MAIN_SIZE))
        {
            printf("[Endurance] Cycle %lu\r\n", cycle);
            printf("[Endurance] Page %lu Read Failed\r\n", page);
            verifyFail = true;
            break;
        }

        for (int i = 0; i < PAGE_MAIN_SIZE; i++)
        {
            if (readBuffer[i] ≠ writeBuffer[i])
            {
                printf("[Endurance] Verify Mismatch at Byte %d \r\n", i);
                printf("[Endurance] Exp : 0x%02X\r\n", writeBuffer[i]);
                printf("[Endurance] Got : 0x%02X\r\n", readBuffer[i]);
                verifyFail = true;
                break;
            }
        }
        if (verifyFail)
            break;
    }
    if (verifyFail)
        break;
```

Endurance Test (Full Block 64 Pages P/E Cycle 60,000)

```
        /// Step 5:
        /// Check ECC and Status Registers (SR1 - SR3)
        /// If P_Fail / E_Fail / ECC Status == ECC_UNCORRECTABLE, End Test
        ECC_Status_t ecc = GetECCStatus_service();
        uint8_t sr1 = GetSR1();
        uint8_t sr2 = GetSR2();
        uint8_t sr3 = GetSR3();

        printf("[Cycle %lu] SR1=0x%02X SR2=0x%02X SR3=0x%02X ECC=%d\r\n", cycle,
                sr1, sr2, sr3, ecc);

        if (CheckEraseFail_service() || CheckProgramFail_service())
        {
            printf("[Cycle %lu] EFAIL/PFAIL Detected\r\n", cycle);
            break;
        }

        if (ecc == ECC_UNCORRECTABLE)
        {
            printf("[Cycle %lu] ECC Uncorrectable → End Test\r\n", cycle);
            break;
        }

        /// Step 6:
        /// Display progress
        if ((cycle + 1) % 100 == 0)
        {
            printf("[Progress] %lu cycles completed\r\n", cycle + 1);
        }
    }

    printf("========== [Endurance Test Finished] ==========\r\n");
    printf("==============================================\r\n");
}
```

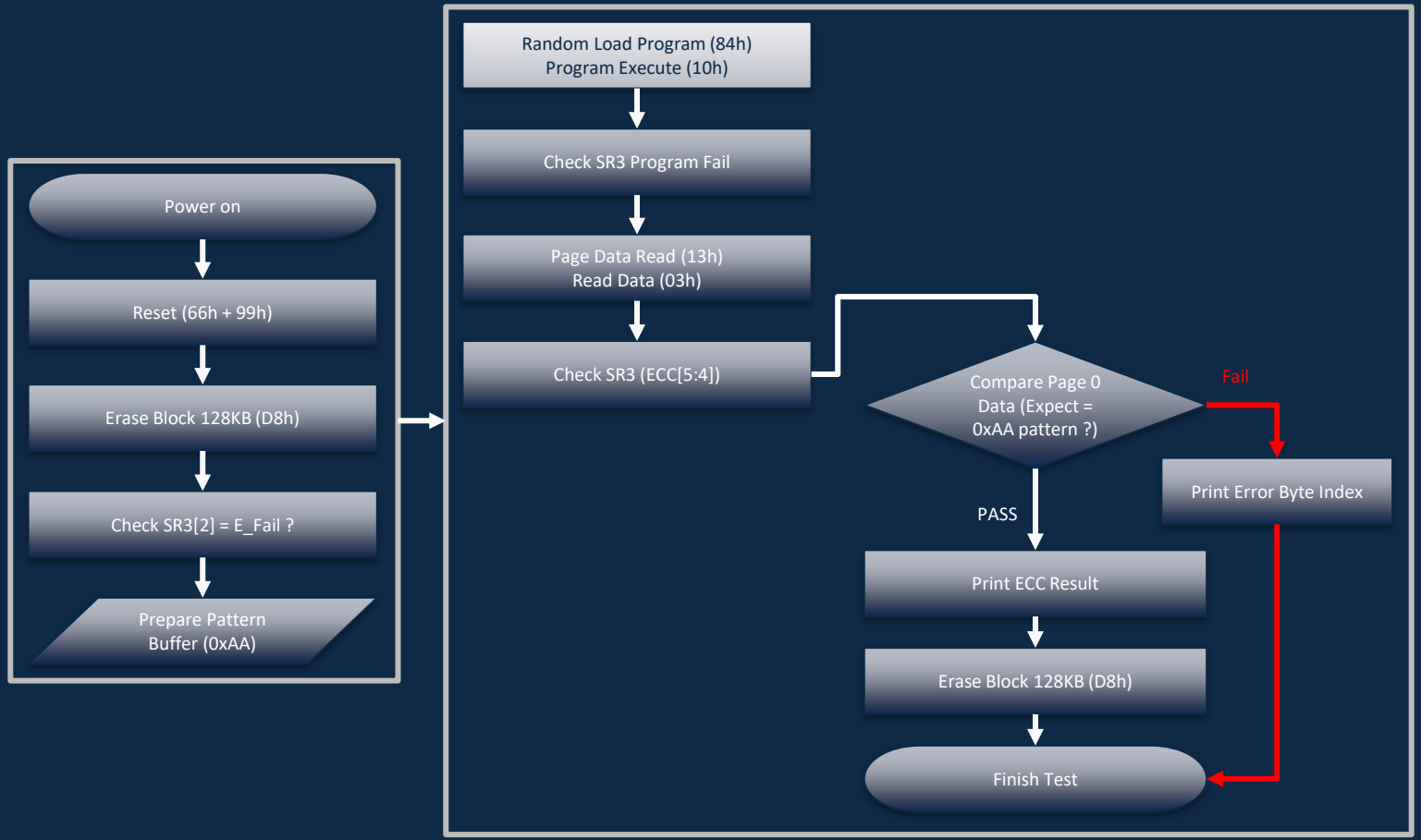
Factory Bad Block Scan



Factory Bad Block Scan

```c
void ScanInvaliBlocks(void)
{
    printf("═══════════════════════════════════════════════════════\r\n");
    printf("═══════════ [Scan Invalid Block Test Start] ═══════════\r\n");

    /// Step 1:
    /// [66h + 99h]: Clear status register and terminate any ongoing operations
    SoftwareReset_service();

    /// Step 2:
    /// Scan Factory Invalid Blocks (8 ~ 2043)
    /// When ECC is enabled, Blocks 0-7 and 2044-2047 are always valid
    /// Invalid block condition: Spare[0] ≠ 0xFF
    BBT_ScanFactoryBlocks();

    /// Step 3: Print Scan Summary
    BBT_PrintSummary();

    /// Step 4: Show Invalid Block List
    BBT_ShowBadBlock();

    printf("═══════════ [Scan Invalid Block Test Finished] ═══════════\r\n");
    printf("═══════════════════════════════════════════════════════\r\n");
}
```

Single Page Program/Read (Random Load Program)



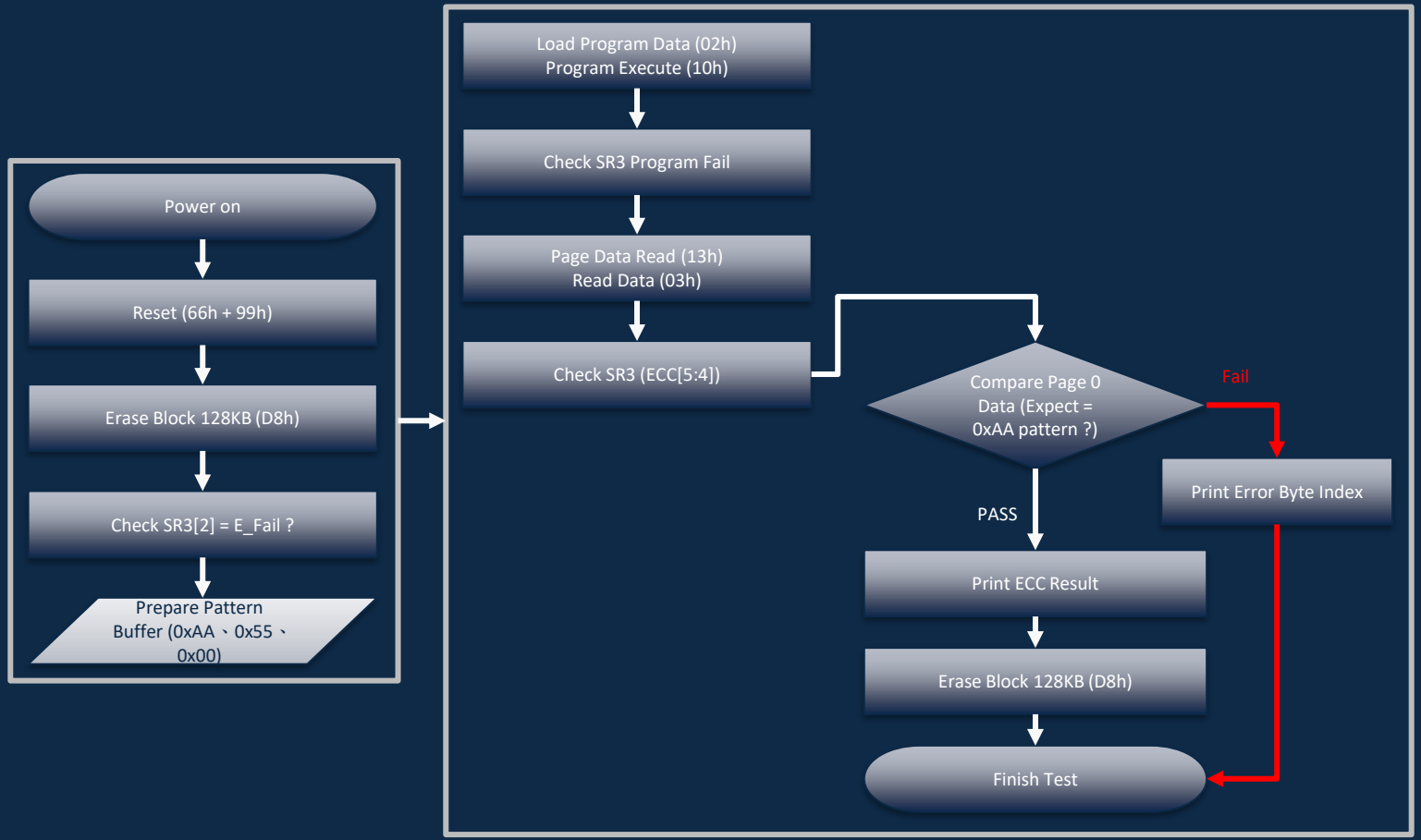
Single Page Read / Program (Random Load Program)

```c
void Random_UnitTest(uint32_t block_num)
{
    uint8_t readBuffer[PAGE_MAIN_SIZE];
    uint8_t writeBuffer[PAGE_MAIN_SIZE];
    uint32_t base_page = block_num * PAGES_PER_BLOCK;

    printf("═══════════════════════════════════════════════════════════\r\n");
    printf("═══════════ [Random UnitTest Start] ═══════════\r\n");

    /// Step 1:
    /// [66h + 99h]: Clear status register and terminate any ongoing operations
    SoftwareReset_service();

    /// Step 2:
    /// [D8h] Block Erase (128KB) → Check Status Register (S2: E_Fail)
    /// Prepare test data: Pattern 0xAA
    BlockErase128K_service(block_num, 500);
    PreparePattern(writeBuffer, PAGE_MAIN_SIZE, PATTERN_AA);

    /// Step 3:
    /// [06h] Write Enable → [84h] Random Load Program Data → [10h] Program Execute
    /// Check Status Register (SR3[3] = P_Fail), return summary when test fail
    RandomProgram_Service(base_page, 0x0000, writeBuffer, PAGE_MAIN_SIZE);
```

Single Page Read / Program (Random Load Program)

```
    /// Step 4:
    /// [13h] Page Data Read → [03h] Read Data
    /// Check ECC Status (SR3[5:4]), return summary when test fail
    RandomRead_Service(base_page, 0x0000, readBuffer, PAGE_MAIN_SIZE);

    for (int i = 0; i < PAGE_MAIN_SIZE; i++)
    {
        if (readBuffer[i] ≠ 0xAA)
        {
            printf("[Verify] Mismatch Byte : %d\r\n", i);
            printf("[Verify] Expect : 0x%02X\r\n", writeBuffer[i]);
            printf("[Verify] Get : 0x%02X\r\n", readBuffer[i]);
            break;
        }
    }

    /// Step 5:
    /// Check Status Register (SR1 - SR3)
    printf("[Status Register] SR1:0x%02X SR2:0x%02X SR3:0x%02X\r\n", GetSR1(),
           GetSR2(), GetSR3());
    printf("[Random UnitTest] Block %lu Test Finished\r\n", block_num);

    /// Step 6:
    /// [D8h] Erase Block again to restore clean state
    BlockErase128K_service(block_num, 500);

    printf("═══════════ [Random UnitTest Finished] ═══════════\r\n");
    printf("═══════════════════════════════════════════════════\r\n");
}
```

Single Page Program / Read (Multiple Pattern)



Single Page Program/Read (Multiple Pattern)

```c
void MultiplePattern_Test(uint32_t block_num)
{
    uint8_t readBuffer[PAGE_MAIN_SIZE];
    uint8_t writeBuffer[PAGE_MAIN_SIZE];
    uint32_t base_page = block_num * PAGES_PER_BLOCK;

    printf("═══════════════════════════════════════════════════════════\r\n");
    printf("═══════════ [MultiplePattern Test Start] ═══════════\r\n");

    /// Step 1:
    /// [66h + 99h]: Clear status register and terminate any ongoing operations
    SoftwareReset_service();

    if (GetSR1() & 0x01)
        printf("[SR1] Busy after Reset\r\n");

    /// Step 2:
    /// [D8h] Block Erase (128KB) → Check Status Register (S2: E_Fail)
    /// Prepare test data: Pattern 0xAA / 0x55 / 0xFF / 0x00
    BlockErase128K_service(block_num, 500);
    PreparePattern(writeBuffer, PAGE_MAIN_SIZE, PATTERN_SEQ_AA55FF00);

    // Step 3:
    /// [06h] Write Enable → [02h] Load Program Data → [10h] Program Execute
    /// Check Status Register (SR3[3] = P_Fail), return summary when test fail
    RandomProgram_Service(base_page, 0x0000, writeBuffer, PAGE_MAIN_SIZE);
```

Single Page Program/Read (Multiple Pattern)

```
    /// Step 4:
    /// [13h] Page Data Read → [03h] Read Data
    /// Check ECC Status (SR3[5:4]), return summary when test fail
    RandomRead_Service(base_page, 0x0000, readBuffer, PAGE_MAIN_SIZE);

    for (int i = 0; i < PAGE_MAIN_SIZE; i++)
    {
        if (readBuffer[i] ≠ writeBuffer[i])
        {
            printf("[Verify] Mismatch Byte : %d\r\n", i);
            printf("[Verify] Expect : 0x%02X\r\n", writeBuffer[i]);
            printf("[Verify] Get : 0x%02X\r\n", readBuffer[i]);
            break;
        }
    }

    /// Step 5:
    /// Check Status Register (SR1 - SR3)
    printf("[SR] SR1 = 0x%02X, SR2 = 0x%02X, SR3 = 0x%02X\r\n", GetSR1(),
           GetSR2(), GetSR3());
    printf("[MultiplePattern Test] Block %lu Test Finished\r\n", block_num);

    /// Step 6:
    /// [D8h] Erase Block again to restore clean state
    BlockErase128K_service(block_num, 500);

    printf("========== [MultiplePattern Test Finished] ==========\r\n");
    printf("=====================================================\r\n");
}
```