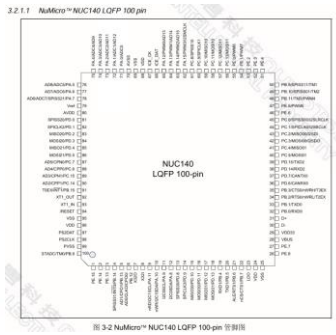
微處理機系統與介面技術 LAB 4

系所：電機 學號 :612415013 姓名：蕭宥羽

<實驗器材>

NUC 140 V2.0 開發板



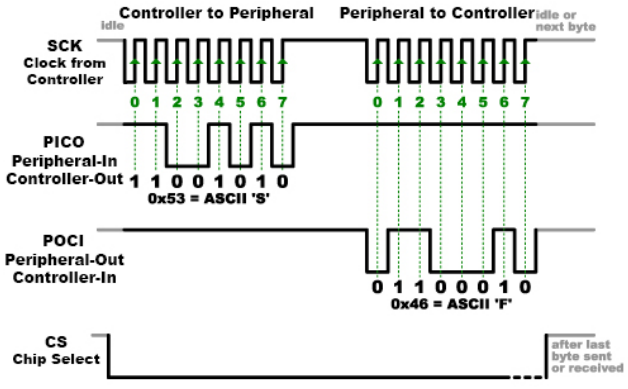
<實驗過程與方法>

- 實驗要求 ：

  使用 SPI Read 3 axis accelerometer and print on putty

  Need to do calibration Result = (Raw data ± offset)/(256 ± offset)

- SPI 原理 ：SPI 是一種同步的串列數據通訊方式，通常可在全雙工模式下運作。
  使用 4 條線進行通訊：SCK（時鐘）、MOSI（主輸出從輸入）、MISO（主輸入從輸出）和 SS（從機選擇）。主機通過 SCK 產生時鐘信號來同步資料的傳輸，MOSI 用於主機向從機發送數據，MISO 則用於從機向主機回傳數據。SS 用於選擇要通訊的特定從機，並控制資料的開始和結束。



- SPI 優缺點
  - ✓ 優點
    - ■ 比非同步串列更快
    - ■ 可以支援多個設備
    - ■ 接收硬體可以是簡單的移位暫存器

- ✓ 缺點
  - 需要多條的信號線
  - 通信必須事先明確定義（不能隨時發送隨機數量的資料）
  - 控制器必須控制所有通訊（外圍裝置不能直接相互通信）
  - 通常需要為每個外設提供單獨的 SS 線，如果需要大量 slaver，這可能會有問題。

NUC140 SPI

本次 lab 使用 SPI2，對應的 GPIO 為 GPD，對應的腳位如下圖

CS ---------->SPI2 CS(GPD0)
SCL ---------->SPI2 CLK(GPD1)
SDO --------->SPI2 MISO(GPD2)
SDA(SDI) ---->SPI2 MOSI(GPD3)

- ✓ SPI register
  - SPI->SSR (從機選擇暫存器)
    - ✧ SS_LVL

      從機選擇訊號的觸發電平，選擇訊號（SS, Slave Select）是用高電平還是低電平來激活 SPI 從機

[2]	**SS_LVL**	**从机选择触发电平** 定义从机选择信号 (SPISSx0/1) 的有效状态. 1 = 从机选择信号 SPISSx0/1在高电平/上升沿有效 0 = 从机选择信号 SPISSx0/1在低电平/下降沿有效

    - ✧ AUTOSS

      如果設為 1 的話，允許主機自動控制從機選擇訊號，當 SPI 控制器設置 GO_BUSY 位元開始傳輸時，從機選擇信號（SPISSx0/1）將會自動變為有效狀態，表示開始與指定的從機通信。傳輸完成後，從機選擇信號自動恢復為無效。

      本次 lab 要將這邊設為 0，手動控制從機選擇訊號。

[3]	**AUTOSS**	**自动从机选择 (Master only)** 1 = 该位置位，SPISSx0/1 信号自动产生。这表示当通过设置 GO_BUSY 位开始发送/接收时，SSR[1:0] 中设定的设备/从机选择信号将由 SPI 控制器设为有效状态，而当每次发送/接收结束时，设备/从机选择信号又会设为无效状态。 0 = 如果该位清零，从机选择信号将由 SSR[1:0] 相关位的设定值来决定激活或失效。

    - ✧ SSR

      從機選擇暫存器，當 AUTOSS 未啟用時，用於手動選擇與主機通信的從機

[1:0]	**SSR**	**从机选择寄存器 (Master only)** 如果 AUTOSS 位被清零，写 1到 SSR[1:0] 任一位将会激活所对应的 SPISSx0/1

  - SPI->CNTRL (控制與狀態暫存器)
    - ✧ SLAVE

      NUC140 可以作為 master or slave，這邊設置 SPI 的操作模式為主機或從機模式

[18]	**SLAVE**	**从机模式指示** 1 = 从机模式 0 = 主机模式

✧ CLKP

決定 SPI 時鐘信號的空閒狀態，如果 CLKP = 1，也就是說 SPI 時鐘在空閒狀態下保持高電平，那麼數據觸發會發生在變成 low 的過程中(下降沿 or 上升沿)

[11]	**CLKP**	**时钟极性** 1 = SPICLK 空闲高电平 0 = SPICLK 空闲低电平

✧ TX_NUM, TX_BIT_LEN

發送/接收的字數與位元長度

➢ TX_NUM : 可以選擇一次傳輸一個或兩個字

[9:8]	**TX_NUM**	**发送/接收的字数目** 该域用于标示一次传输中，发送/接收的字数目。 00 = 一次传输仅发送/接收一个字
		01 = 一次传输发送/接收两个连续的字 (burst mode) 10 = 保留 11 = 保留 注：从机电平触发模式时，若 TX_NUM 设置为01，从机选择管教信号在连续的数据传输过程中，必须保持有效状态。

➢ TX_BIT_LEN : 指定 SPI 傳輸的位元數，可以傳輸不同位元長度的數據

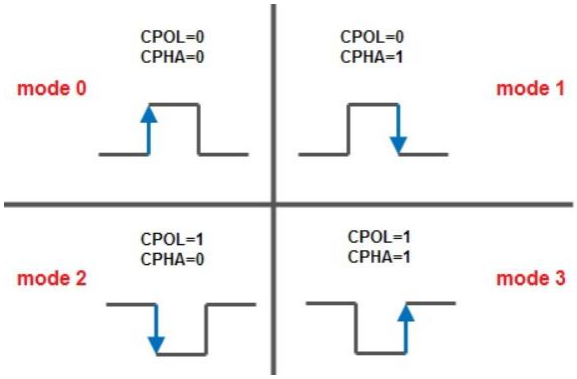
[7:3]	**TX_BIT_LEN**	**传输位长度** 该域用于标示一次传输中，完成的传输比特位，最高为32位 TX_BIT_LEN = 0x01 … 1 bit TX_BIT_LEN = 0x02 … 2 bits …… TX_BIT_LEN = 0x1F … 31 bits TX_BIT_LEN = 0x00 … 32 bits

✧ TX_NEG, RX_NEG

SPI 在傳輸和接收數據時是使用時鐘的上升沿還是下降沿。

[2]	**TX_NEG**	**发送数据边沿反向位** 1 = 在 SPICLK 下降沿改变发送数据输出信号 0 = 在 SPICLK 上升沿改变发送数据输出信号
[1]	**RX_NEG**	**负边沿接收** 1 = 在 SPICLK 的下降沿锁存接收数据输入信号 0 = 在 SPICLK 的上升沿锁存接收数据输入信号

像是下圖這樣，選擇觸發方式



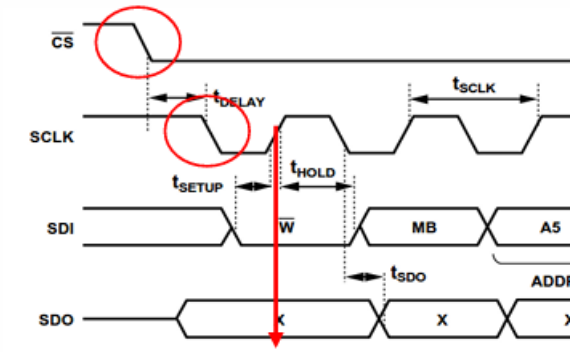
✧ GO_BUSY

控制 SPI 的啟動和傳輸過程，設定為 1 會開始傳輸，當傳輸完成後自動清零。

[0]	GO_BUSY	通讯或忙状态标志 1 = 在主机模式下，写 1 到该位开始 SPI 数据传输；在从机模式下，写 1 到位表示从机准备好与主机进行通信。 0 = 当 SPI 在传输时，写 0 到该位停止数据传输。 数据传输过程中，该位值保持为 1，当传输完成后，该位自动清零。 注： 1. 在写 1 到 GO_BUSY 位之前，所有的寄存器必须设置完成。

- ADXL345 SPI
  - ✓ NUC140 SPI 設置

    MCU SPI 的設定要根據 ADXL 的接收方式去做設定，由下面的圖(ADXL data sheet)可以看出
    - ss/cs 拉低會啟動 SPI 通信 ： SS_LVL 要設成 0 (低電平有效)
    - NUC140 為 master： SLAVE = 0 (master mode)
    - 時鐘空閒時為高電平 ：CPOL = 1，設置 CLKP = 1
    - 時鐘的上升沿觸發 ：CPHA = 1，設置 TX_NEG, RX_NEG = 0 (mode3)
    - TX_BIT_LEN(8 bit data length for each word transmit) : TX_BIT_LEN=0x08
    - TX_NUM(One word in one transfer): TX_NUM=00 一次傳輸一個字



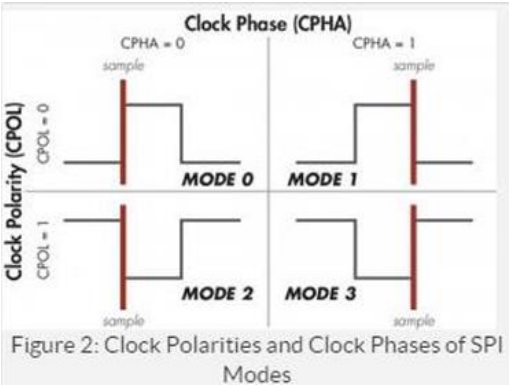


Figure 2: Clock Polarities and Clock Phases of SPI Modes

  - ✓ SPI 4-Wire Write
    1. CS 拉低，SPI 開始通信
    2. 主機通過 SDI 發送控制位和地址位，表示要寫入的暫存器
       - ✧ W/R (bit7) : write(0) read(1)
       - ✧ MB (bit6) :

         MB = 1，表示將連續讀取或寫入多個位元組(連續的暫存器地址)

         MB = 0，表示僅進行單位元組的讀取或寫入
       - ✧ Address (bit0-bit5)
    3. 接著主機發送數據位（D7-D0），要寫入到指定暫存器的數據
    4. CS 拉高， SPI 傳輸結束

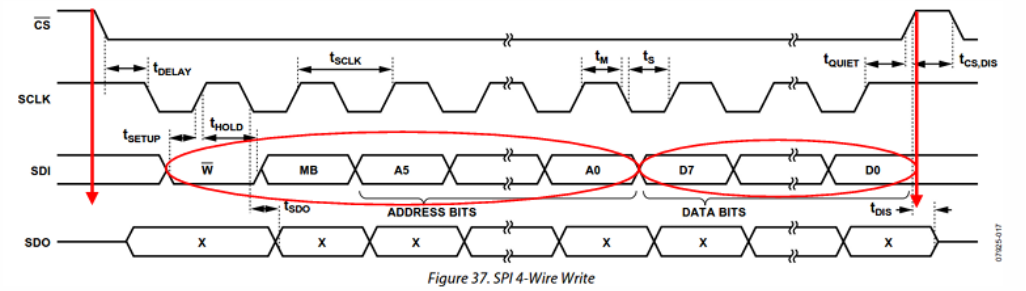


Figure 37. SPI 4-Wire Write

✓ SPI 4-Wire Read

1,2,4 同 SPI 4-Wire Write

3. 從機根據接收到的地址位讀取相應的寄存器，並通過 SDO 發送數據位（D7-D0）給主機

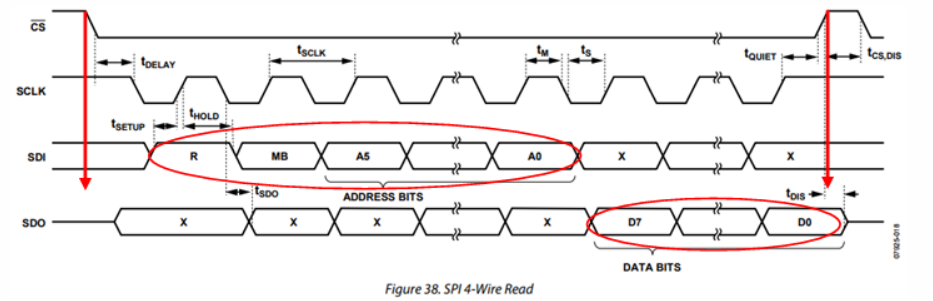


Figure 38. SPI 4-Wire Read

✓ Initial ADXL345

■ POWER_CTL(0x2D): 0x08

D3 = 1，其餘位元為 0，啟用測量模式

Register 0x2D—POWER_CTL (Read/Write)

Table 26. Register 0x2D

D7	D6	D5	D4	D3	D2	D1	D0
0	0	Link	AUTO_SLEEP	Measure	Sleep	Wakeup	

■ DATA_FORMAT(0x31): 0x0B(0000_1011)

Register 0x31—DATA_FORMAT (Read/Write)

Table 34. Register 0x31

D7	D6	D5	D4	D3	D2	D1	D0
SELF_TEST	SPI	INT_INVERT	0	FULL_RES	Justify	Range	

Table 35. g Range Setting

Setting		
D1	D0	g Range
0	0	±2 g
0	1	±4 g
1	0	±8 g
1	1	±16 g

■ FIFO_CTL(0x38): 0x80

D7-D6（FIFO_MODE）：設定 FIFO 的工作模式()

✧ STREAM mode(D7-D6= 10)

當 FIFO 緩衝器滿了時，最舊的數據會被丟棄，以便保存新的數據。因此，FIFO 始終保存最新的數據樣本。

Register 0x38—FIFO_CTL (Read/Write)

Table 36. Register 0x38

D7	D6	D5	D4	D3	D2	D1	D0
FIFO_MODE		Trigger	Samples				

Table 37. FIFO Modes

Setting			
D7	D6	Mode	Function
0	0	Bypass	FIFO is bypassed.
0	1	FIFO	FIFO collects up to 32 values and then stops collecting data, collecting new data only when FIFO is not full.
1	0	Stream	FIFO holds the last 32 data values. When FIFO is full, the oldest data is overwritten with newer data.
1	1	Trigger	When triggered by the trigger bit, FIFO holds the last data samples before the trigger event and then continues to collect data until full. New data is collected only when FIFO is not full.

✓ ADXL data register

ADXL345 的加速度數據是由 16 位元組成的，每個軸的數據分為兩個 8 位暫存器存儲

DATAX0(0x32), DATAX1(0x33)
DATAY0(0x34), DATAY1(0x35)
DATAZ0(0x36), DATAZ1(0x37)

- <Mian function code>
  - ✓ 會使用以下這些 function

```
void SYS_Init(void);
void SPI_Init(void);

void ADXL_write(uint8_t address, uint8_t data);
uint8_t ADXL_read(uint8_t address);
void ADXL_init(void);
```

1. SYS_Init

   PD.0 設置為 SPI2 SS0，用於選擇從機。

   PD.1 設置為 SPI2 SPICLK，用於 SPI2 的時鐘信號。

   PD.2 設置為 SPI2 MISO0，用於數據的輸入（從機到主機）。

   PD.3 設置為 SPI2 MOSI0，用於數據的輸出（主機到從機）。

```
/* Setup SPI2 multi-function pins */  //GPD_MFP  page.80
SYS->GPD_MFP = SYS_GPD_MFP_PD0_SPI2_SS0 | SYS_GPD_MFP_PD1_SPI2_CLK | SYS_GPD_MFP_PD2_SPI2_MISO0 | SYS_GPD_MFP_PD3_SPI2_MOSI0;
```

[3]	**GPD_MFP3**	**PD.3 管脚功能选择** 1 = PD.3 作为 SPI2 MOSI0（主机输出，从机输入管脚-0） 0 = PD.3 作为 GPIOD[3]
[2]	**GPD_MFP2**	**PD.2 管脚功能选择** 1 = PD.2 作为 SPI2 MISO0（主机输入，从机输入管脚-0） 0 = PD.2 作为 GPIOD[2]
[1]	**GPD_MFP1**	**PD.1 管脚功能选择** 1 = PD.1 作为 SPI2 SPICLK 0 = PD.1 作为 GPIOD[1]
[0]	**GPD_MFP0**	**PD.0 管脚功能选择** 1 = PD.0 作为 SPI2 SS20 0 = PD.0 作为 GPIOD[0]

2. SPI_Init

   - SPI_Open(SPI2, SPI_MASTER, SPI_MODE_3, 8, 2000000);
     - ✧ SPI2：要初始化的 SPI 模組(SPI2)
     - ✧ SPI_MASTER：設置 SPI2 為主機模式（Master）
     - ✧ SPI_MODE_3 : CPOL = 1, CPHA = 1
     - ✧ 8：數據長度為 8 位元
     - ✧ 2000000：SPI 的時鐘頻率

```
void SPI_Init(void)
{
    SPI_Open(SPI2, SPI_MASTER, SPI_MODE_3, 8, 2000000);
    SPI_DisableAutoSS(SPI2);
}
```

   - SPI_Open 實際操作方式是如下圖，他會對 spi->CNTRL 去做寫入，完成 SPI 的初始設定

```
uint32_t SPI_Open(SPI_T *spi,
                  uint32_t u32MasterSlave,
                  uint32_t u32SPIMode,
                  uint32_t u32DataWidth,
                  uint32_t u32BusClock)
```

```
/* Default setting: MSB first, disable unit transfer interrupt, SP_CYCLE = 0. */
spi->CNTRL = u32MasterSlave | (u32DataWidth << SPI_CNTRL_TX_BIT_LEN_Pos) | (u32SPIMode);
```

3. ADXL_write

這邊就是 ADXL 寫入的流程，MCU 藉由 SPI 傳遞暫存器位置給 ADXL，接這再傳送要寫入的內容，對 ADXL 暫存器做寫入

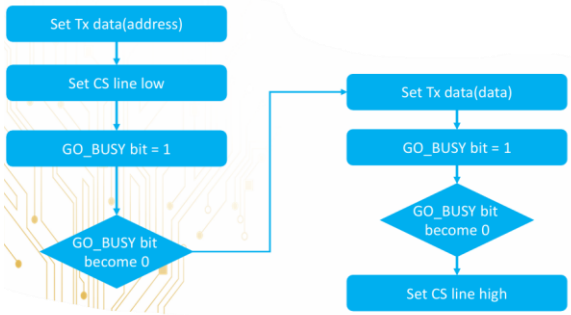
- SPI_WRITE_TX0(SPI2, 0x3F & address);

  0x3F & address：這裡將地址與 0x3F 進行按位與操作，讓最高位為 0，做寫操作
- SPI_SET_SS0_LOW(SPI2); 拉低從機選擇信號，開始通信
- SPI_TRIGGER(SPI2); while (SPI_IS_BUSY(SPI2)); 觸發 SPI 傳輸並等待完成
- 接這做同樣的流程寫入數據
- SPI_SET_SS0_HIGH(SPI2); 將 SS0 拉高，表示此次通信結束

```
void ADXL_write(uint8_t address, uint8_t data){

    SPI_WRITE_TX0(SPI2, 0x3F&address);
    SPI_SET_SS0_LOW(SPI2);
    SPI_TRIGGER(SPI2);
    while(SPI_IS_BUSY(SPI2));

    SPI_WRITE_TX0(SPI2, data);
    SPI_TRIGGER(SPI2);
    while(SPI_IS_BUSY(SPI2));
    SPI_SET_SS0_HIGH(SPI2);
}
```



4. ADXL_read

這邊跟 write 很類似，MCU 藉由 SPI 傳遞暫存器位置給 ADXL，決定我要讀取哪個 ADXL 的暫存器，接著 ADXL 用 SPI 將暫存器內容傳給 MCU

- SPI_WRITE_TX0(SPI2, 0x80|address);

  0x80|address：這裡將地址與 0x80 進行按位與操作，讓最高位為 1，做讀操作

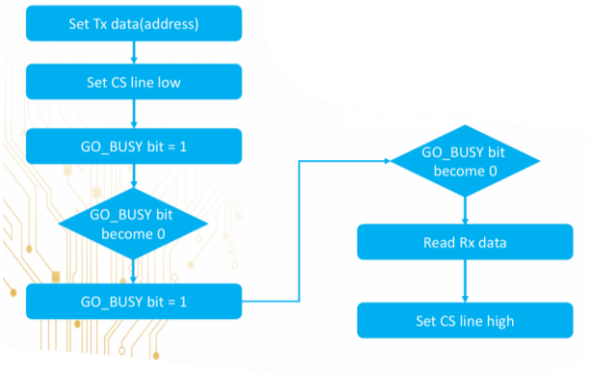
```
uint8_t ADXL_read(uint8_t address){

    uint8_t read_content;
    SPI_WRITE_TX0(SPI2, 0x80|address);
    SPI_SET_SS0_LOW(SPI2);
    SPI_TRIGGER(SPI2);
    while(SPI_IS_BUSY(SPI2));

    //Read the contents of register
    SPI_TRIGGER(SPI2);
    while(SPI_IS_BUSY(SPI2));
    read_content = SPI_READ_RX0(SPI2);

    SPI_SET_SS0_HIGH(SPI2);

    return read_content;
}
```



5. ADXL_init

對 ADXL 的這些暫存器寫入這些值 (上面有說這些暫存器的作用)

```
void ADXL_init(void){

    ADXL_write(0x2D, 0x08);
    ADXL_write(0x31, 0x0B);
    ADXL_write(0x38, 0x80);
}
```

6. Main

- 先讀取 ADXL 0x00 的內容，讀到的值一定要是 0xE5，我用這邊檢視 SPI 的傳輸程式碼是否是正常的

Hex	Dec	Name	Type	Reset Value	Description
0x00	0	DEVID	R	11100101	Device ID

```
device_num = ADXL_read(0x00);
printf("\ndevice: 0x%02X\n", device_num);
```

- while(1)
  - rd_xaxis = (ADXL_read(0x33) << 8) | ADXL_read(0x32);
    - **ADXL345** 的加速度數據是 **16 位元**，分為高位和低位來存儲在暫存器中
    - 先讀取高位（0x33），並左移 8 位，再與低位（0x32）進行按位 OR，以獲得完整的 **16 位 X 軸加速度數據**。
    - Y Z 是同樣的概念
  - 偏移補償

    ADXL_read(0x1E)、ADXL_read(0x1F)、ADXL_read(0x20) 分別讀取 X、Y、Z 三個軸的偏移暫存器（偏移補償）的值，用於補償加速度感測器的零點漂移。

    Register 0x1E, Register 0x1F, Register 0x20—OFSX, OFSY, OFSZ (Read/Write)

    The OFSX, OFSY, and OFSZ registers are each eight bits and offer user-set offset adjustments in twos complement format with a scale factor of 15.6 mg/LSB (that is, 0x7F = 2 *g*). The value stored in the offset registers is automatically added to the acceleration data, and the resulting value is stored in the output data registers. For additional information regarding offset calibration and the use of the offset registers, refer to the Offset Calibration section.

  - 最後用 offset 進行校正，得到最終 x y z 的值

$$\text{Result} = (\text{Raw data} \pm \text{offset})/(256 \pm \text{offset})$$

```
while(1)
{

  rd_xaxis = (ADXL_read(0x33)<<8)|ADXL_read(0x32);
  rd_yaxis = (ADXL_read(0x35)<<8)|ADXL_read(0x34);
  rd_zaxis = (ADXL_read(0x37)<<8)|ADXL_read(0x36);


  offset = ADXL_read(0x1E);
  cd_xaxis = (float)(rd_xaxis - offset) / (256 - offset);


  offset = ADXL_read(0x1F);
  cd_yaxis = (float)(rd_yaxis - offset) / (256 - offset);

  offset = ADXL_read(0x20);
  cd_zaxis = (float)(rd_zaxis - offset) / (256 - offset);

  printf("x: %.2f, y: %.2f, z: %.2f\n", cd_xaxis, cd_yaxis, cd_zaxis);

  CLK_SysTickDelay(500000);  //delay 500ms

}
```

<心得與收穫>

這次的 SPI 實驗看似簡單，但在實作過程中發現，成功實現 SPI 功能，需要對 SPI 通訊協議及其硬體配置有深入的理解。每一個步驟都需要仔細設定，例如 SPI 的主從模式選擇、時鐘極性和相位的配置（CPOL、CPHA），以及從機選擇信號（SS）的控制，都需要對 SPI 模組的工作機制及硬體設計有充分的認識。此外，如何正確初始化 SPI，管理數據的傳輸，以及確保通訊的正確性，這些細節都非常重要。

在實驗過程中，我學習到如何有效設置 SPI 的參數，包括主從模式的選擇、數據位寬的設置，

以及如何確保通訊時序與數據取樣的正確性。我理解了 SPI 模式（Mode 0-3）對數據傳輸的影響，並且明白了如何配置從機選擇信號以實現對不同設備的控制。這些操作看似簡單，但對於精確控制 SPI 通訊過程以及確保從機正確接收數據都非常重要。

同時，我也體會到在 SPI 實作中，系統初始化和數據傳輸的時序管理是非常關鍵的部分。從 SPI 控制寄存器的配置，到拉低並釋放從機選擇信號，每一個步驟都要求系統性和精確度，以確保數據的正確性並避免在通訊過程中的干擾與錯誤。

這次實驗給了我寶貴的經驗，使我對嵌入式系統中的 SPI 模組有了更深刻的理解，也明白了如何在硬體和軟體之間協調，以實現穩定、準確的 SPI 通訊。