CHƯƠNG 3: LẬP TRÌNH KIT STM32

3.1. Lập trình qua các chân vào ra chung

Các bộ vi xử lý STM32 là những hệ thống phức tạp với nhiều thiết bị ngoại vi. Bất kỳ thiết bị ngoại vi cũng phải được cấu hình trước khi sử dụng. Một số cấu hình này là dung chung – chẳng hạn cấu hình xung clock hay chân, phần còn lại là cấu hình thiết bị ngoại vi cụ thể. Trong phần này chúng ta sẽ xây dựng chương trình nháy LED làm thí dụ cụ thể.

Các bước khởi tạo cơ bản cần thiết để sử dụng bất kỳ thiết bị ngoại vi nào của STM32 là:

1. Kích hoạt xung clock cho ngoại vi
2. Cấu hình chân theo yêu cầu của các thiết bị ngoại vi
3. Cấu hình phần cứng của ngoại vi

Vi xử lý STM32 cũng như các thành viên của họ Cortex-M3 đều có một bộ định thời hệ thống được sử dụng để cung cấp xung nhịp thường xuyên "tick". Thí dụ sau sử dụng timer này để tạo tốc độ nháy liên tục cho các LED. Cấu trúc tổng thể của chương trình này được minh họa trong hình dưới đây. Chương trình bắt đầu bằng việc khao báo các thư viện phần mềm có liên quan - trong trường hợp này là khai báo timer hệ thống và cấu hình các pin. Chương trình chính gồm các bước khởi tạo được mô tả ở trên và một vòng lặp chứa hàm Toggles() bật một đèn LED sang và chờ 250ms.

Hàm tiếp theo thực hiện chức năng tạo trễ trong đó sử dụng timer hệ thống. Cuối cùng, một một hàm trợ giúp được cung cấp để xử lý vi phạm cho phép trong thư viện firmware (cần thiết lệnh if USE_FULL_ASSERT được định nghĩa khi biên dịch các module trong thư viện firmware). Nội dung hàm assert_failed không làm gì nhưng nó là rất hữu ích khi gỡ lỗi các dự án khi các thư viện phần mềm sẽ thực hiện kiểm tra tham số mở rộng. Trong trường hợp vi phạm vùng quản lý,

trình debug GDB có thể được sử dụng để kiểm tra các thông số của hàm này để xác định điểm thất bại.

```
#include <stm32f10x.h>
#include <stm32f10x_rcc.h>
 #include <stm32f10x_gpio.h>
void Delay(uint32_t nTime);
int main(void) {
        GPIO_InitTypeDef GPIO_InitStructure;
        // Enable Peripheral Clocks
        ... (1) ...
        // Configure Pins
        ... (2) ...
        // Configure SysTick Timer
        ... (3) ... while (1) {
        static int ledval = 0;
        // toggle led
        ... (4) ...
        Delay(250); // wait 250ms }
        }
// Timer code
... (5) ...
#ifdef USE_FULL_ASSERT
void assert_failed(uint8_t* file, uint32_t line) {
        /* Infinite loop */
        /* Use GDB to find out why we're here */ while (1);
} #endif
```

KIT STM32 VL discovery có một LED được ghép nối tại chân PC9, chúng ta phải cấu hình xung Clock như sau:

RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE); // (1)

Trong thế giới của các bộ xử lý nhúng, điện năng tiêu thụ là rất quan trọng; do đó, bộ xử lý nhúng tinh vi nhất cung cấp cơ chế tắt nguồn khi không cần thiết cho một ứng dụng cụ thể. STM32 có một mạng lưới phân phối xung clock phức tạp để đảm bảo rằng chỉ những thiết bị ngoại vi thực sự cần thiết được cung cấp. Hệ thống này, được gọi là Reset và Clock Control (RCC) được hỗ trợ bởi các mô-đun firmware stm32f10x_rcc. [ch]. Trong khi mô-đun này có thể được sử dụng để kiểm soát các xung clock hệ thống và các PLLs chính, bất kỳ yêu cầu cấu hình nào của người lập trình cũng được xử lý bởi phần mã khởi động. Ở đây chúng ta đơn giản chỉ kích hoạt xung clock cho các ngoại vi.

Các thiết bị ngoại vi của STM32 được tổ chức thành ba nhóm riêng biệt gọi là APB1, APB2, và AHB. Thiết bị ngoại vi APB1 bao gồm các thiết bị I2C, USART từ 2-5, và các thiết bị SPI; Thiết bị APB2 bao gồm các cổng GPIO, bộ điều khiển ADC và USART1; thiết bị AHB chủ yếu bao gồm các bộ điều khiển DMA và giao diện bộ nhớ ngoài (và một số thiết bị khác).

Xung clock cung cấp cho các thiết bị ngoại vi khác nhau có thể được kiểm soát với một trong ba hàm khởi tạo firmware như sau:

RCC_APB1PeriphClockCmd(uint32_t RCC_APB1PERIPH, FunctionalState NewState)

RCC_APB2PeriphClockCmd(uint32_t RCC_APB2PERIPH, FunctionalState NewState)

RCC_AHBPeriphClockCmduint32_t RCC_AHBPERIPH, FunctionalState NewState)

Mỗi hàm có 2 tham số: một tham số chỉ thiết bị ngoại vi muốn sử dụng và một tham số chỉ hành động sẽ thực hiện với một trong 2 giá trị ENABLE hoặc DISABLE. Thí dụ, để khởi tạo xung clock cho cổng A và cổng B ta có lệnh sau:

RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB, ENABLE);

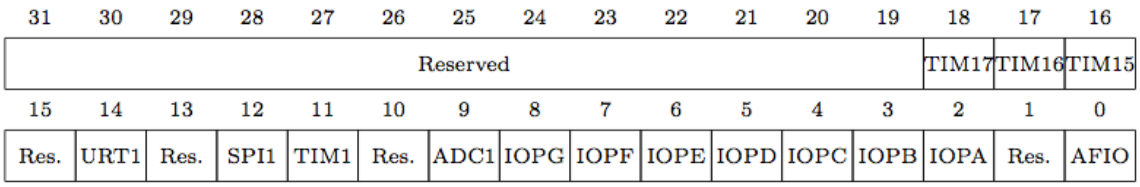
Thí dụ trên là phù hợp với STM32 F1, đối với các STM32 khác phải đọc file dạng stm32fxxxx_rcc.h để xem các định nghĩa xung trên từng bus cụ thể.

APB1 Devices	APB2 Devices
RCC_APB1Periph_BKP	RCC_APB2Periph_ADC1
RCC_APB1Periph_CEC	RCC_APB2Periph_AFIO
RCC_APB1Periph_DAC	RCC_APB2Periph_GPIOA
RCC_APB1Periph_I2C1	RCC_APB2Periph_GPIOB
RCC_APB1Periph_I2C2	RCC_APB2Periph_GPIOC
RCC_APB1Periph_PWR	RCC_APB2Periph_GPIOD
RCC_APB1Periph_SPI2	RCC_APB2Periph_GPIOE
RCC_APB1Periph_TIM2	RCC_APB2Periph_SPI1
RCC_APB1Periph_TIM3	RCC_APB2Periph_TIM1
RCC_APB1Periph_TIM4	RCC_APB2Periph_TIM15
RCC_APB1Periph_TIM5	RCC_APB2Periph_TIM16
RCC_APB1Periph_TIM6	RCC_APB2Periph_TIM17
RCC_APB1Periph_TIM7	RCC_APB2Periph_USART1
RCC_APB1Periph_USART2	
RCC_APB1Periph_USART3	
RCC_APB1Periph_WWDG	
AHB Devices	
RCC_AHBPeriph_CRC	RCC_AHBPeriph_DMA

Bảng 3.1. Bảng phân phối các xung clock trên kít STM32F1

Bảng định nghĩa phân bố các xung clock cho các ngoại vi trên STM32 F1 (stm32f10x_rcc.h)

Ngoài ra, người lập trình có thể cấu hình thanh ghi RCC_APB2ENR để cho phép hoặc không cho phép các xung clock đến các ngoại vi tương ứng. Thanh ghi RCC_APB2ENR gồm các bít cấu hình xung clock như sau:



Hình 3.1. Thanh ghi RCC_APB2ENR với các bít cấu hình xung clock

Thí dụ, để cấu hình xung của cổng GPIOA và GPIOB ta có thể sử dụng lệnh:

ABP2ENR |= 0x0C;

Sau khi cấu hình xung clock cho ngoại vi, chúng ta cấu hình các chân vào ra, trong thí dụ sau ta sẽ cấu hình chân PC9:

```
/* (2) */
GPIO_StructInit(&GPIO_InitStructure);
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(GPIOC, &GPIO_InitStructure);
```

Sau khi được cấu hình xong, chân vào/ra có thể được thiết lập (set) hoặc xoá (reset) nhờ lệnh sau:

```
/* (4) */
GPIO_WriteBit(GPIOC, GPIO_Pin_9, (ledval) ? Bit_SET : Bit_RESET);
ledval = 1 - ledval;
```

Hầu hết các chân của STM32 có thể được cấu hình như là chân đầu vào hoặc đầu ra và có thể được một trong hai chức năng là cổng GPIO hay "chức năng thay thế" (đối với thiết bị ngoại vi khác). Như một quy ước đặt tên chuẩn, các chân được gọi bằng chức năng GPIO của chúng - ví dụ PA0 (bit 0 của cổng A) hoặc PB9 (bit 9 của cổng B).

Với mỗi ràng buộc phần cứng cụ thể, mỗi pin có thể được cấu hình ở một trong các chế độ như sau:

Function	Library Constant
Alternate function open-drain	`GPIO_Mode_AF_OD`
Alternate function push-pull	`GPIO_Mode_AF_PP`
Analog	`GPIO_Mode_AIN`
Input floating	`GPIO_Mode_IN_FLOATING`
Input pull-down	`GPIO_Mode_IPD`
Input pull-up	`GPIO_Mode_IPU`
Output open-drain	`GPIO_Mode_Out_OD`
Output push-pull	`GPIO_Mode_Out_PP`

Bảng 3.2. Các chế độ của một chân (stm32f10x_gpio.h)

Mặc định khi ở chế độ reset tất cả các chân ở chế độ “Input Floating” để đảm bảo không có xung đột phần cứng nào xảy ra. Thư viện firmware cung cấp các hàm khởi tạo trong file stm32f10x_gpio.[ch] để cho phép cấu hình các chân.

Khi cấu hình với đầu ra như trên, chúng ta có ba lựa chọn tốc độ đầu ra là: 50 MHz, 10 MHz, và 2 MHz. Nói chung, vì lý do tiêu thụ điện năng và nhiễu, người ta thường sử dụng các tốc độ thấp khi cấu hình chân với mục đích vào ra.

Để cấu hình các chân kết hợp với các nút nhấn trên KIT, ta sử dụng đoạn code sau:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

Để đọc giá trị hiện hành của nút nhấn ta sử dụng lệnh:

```
GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)
```

16 bít của mỗi cổng vào ra có thể được cấu hình với thanh ghi CRL (các chân từ 0-7) và CHR (chân từ 8-15). Để hỗ trợ các chế độ vào ra khác nhau người ta sử dụng 4 bít cấu hình cho từng chấn của cổng GPIO. Để đọc các bít song song trên cổng sử dụng thanh ghi IDR và ghi song song sử dụng thanh ghi ODR. Các thanh ghi BSRR và BRR cho phép set và reset các bít riêng rẽ. Thanh ghi LCKR cung cấp một cơ chế cho phép khoá thanh ghi để bảo vệ phần cứng khỏi những lỗi khi phần mềm cấu hình lại thanh ghi.

Hằng số SystemCoreClock là số chu kỳ xung clock/giây được định nghĩa trong thư viện ngoại vi:

```
/* (3) */
    if (SysTick_Config(SystemCoreClock / 1000)) while (1);
```

Ở đây, cứ sau msec thì bộ tạo timer lại gọi phương thức SysTick_Handler. Để demo led nhấp nháy, chúng ta giảm bộ đếm chia sẻ bằng cách khao báo __IO để đảm bảo rằng các trình biên dịch không thực hiện các tối ưu không mong muốn.

```
/* (5) */
static __IO uint32_t TimingDelay;
void Delay(uint32_t nTime){
TimingDelay = nTime;
while(TimingDelay != 0);
}
void SysTick_Handler(void){
if (TimingDelay != 0x00)
TimingDelay--; }
```

3.2. Lập trình ngắt ngoài

Phần này sẽ hướng dẫn cách lập trình ngắt ngoài trên kít STM32F4. Mỗi thiết bị STM32F4 có 23 nguồn ngắt ngoài. Chúng được chia làm 2 loại. Loại 1 là các ngoài với các chân từ chân P0 đến chân P15 trên từng cổng và loại 2 là ngắt cho các loại khác như RTC, Ethernet, USB, ….

3.2.1. Ngắt trên các chân vào ra

Các đường ngắt: Chúng ta có thể cấu hình các chân vào ra là các chân ngắt và điều khiển chúng với các hành CMSIS. Ngắt loại 1 có 16 đường ngắt từ line0 đến line15 tương ứng với các chân từ pin0 đến pin15 trên mỗi cổng. Tức là PA0 tương ứng với line0, PA12 tương ứng với line12,….

Tất cả các chân trên kít đều là chân ngắt, Px0 kết nối tới line0, px3 kết nối tới line3 và cứ tiếp tục như vậy. Số hiệu chân cùng với số hiệu line, nhưng tại một thời điểm chỉ có 1 chân kết nối tới 1 line.

Mỗi chân có thể bắt các tín hiệu ngắt khi có tín hiệu sườn tăng hoặc sườn giảm hoặc lúc tăng, lúc giảm.

3.2.2. Điều khiển ngắt:

STM32F4 có 7 trình điều khiển ngắt cho các chân GPIO. Chúng được mô tả trong bảng dưới dây:

Số hiệu ngắt	Tên trình điều khiển	Mô tả
EXTI0_IRQn	EXTI0_IRQHandler	Điều khiển ngắt cho các chân kết nối tới line 0
EXTI1_IRQn	EXTI1_IRQHandler	Điều khiển ngắt cho các chân kết nối tới line 1
EXTI2_IRQn	EXTI2_IRQHandler	Điều khiển ngắt cho các chân kết nối tới line 2
EXTI3_IRQn	EXTI3_IRQHandler	Điều khiển ngắt cho các chân kết nối tới line 3
EXTI4_IRQn	EXTI4_IRQHandler	Điều khiển ngắt cho các chân kết nối tới line 4
EXTI9_5_IRQn	EXTI9_5_IRQHandler	Điều khiển ngắt cho các chân kết nối tới các line từ 5 đến 9
EXTI15_10_IRQn	EXTI15_10_IRQHandler	Điều khiển ngắt cho các chân kết nối tới các line từ 10 đến 15

Bảng 3.3. Trình điều khiển ngắt ngoài cho GPIO trên kít STM32F4

Sau khi thiết lập ngắt, người lập trình có thể thêm nó vào bộ điều khiển ngắt NVIC.

3.3.3. Thí dụ

Chương trình sau sẽ thiết lập các chân PD0 và PB12 là các chân ngắt.

```
#include "stm32f4xx.h"
#include "stm32f4xx_exti.h"
#include "stm32f4xx_syscfg.h"
#include "misc.h"

/* Cấu hình các chân là chân ngắt */
```

```
void Configure_PD0(void) {
/* Khai báo các biến */
GPIO_InitTypeDef GPIO_InitStruct;
EXTI_InitTypeDef EXTI_InitStruct;
NVIC_InitTypeDef NVIC_InitStruct;

/* Cho phép xung clock tại cổng GPIOD */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
/* Cho phép xung clock SYSCFG */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

/* Thiết lập các chân là đầu vào */
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN;
GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
GPIO_InitStruct.GPIO_Pin = GPIO_Pin_0;
GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_Init(GPIOD, &GPIO_InitStruct);

/* Báo cho hệ thống biết chân PD0 sử dụng cho EXTI_Line0 */
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOD, EXTI_PinSource0);

/* PD0 được kết nối tới EXTI_Line0 */
EXTI_InitStruct.EXTI_Line = EXTI_Line0;
/* Cho phép ngắt */
EXTI_InitStruct.EXTI_LineCmd = ENABLE;
/* Chọn chế độ ngắt */
EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
/* Ngắt xảy ra khi có cả sường tăng hoặc giảm */
```

```
EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Rising_Falling;
/* Thêm vào cấu trúc ngắt EXTI */
EXTI_Init(&EXTI_InitStruct);

/* Thêm vector IRQ vào NVIC */
/* PD0 kết nối tới EXTI_Line0 tương ứng với vector EXTI0_IRQn */
NVIC_InitStruct.NVIC_IRQChannel = EXTI0_IRQn;
/* Thiết lập độ ưu tiên */
NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0x00;
/* Thiết lập độ ưu tiên trong nhóm */
NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0x00;
/* Cho phép ngắt*/
NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
/* Thêm vào NVIC */
NVIC_Init(&NVIC_InitStruct);
}

void Configure_PB12(void) {
/* Khai báo các biên */
GPIO_InitTypeDef GPIO_InitStruct;
EXTI_InitTypeDef EXTI_InitStruct;
NVIC_InitTypeDef NVIC_InitStruct;

/* Cho phép xung clock cổng GPIOB */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
/* Cho phép xung clock SYSCFG */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

/* Thiết lập chân là đầu vào */
```

```
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN;
GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
GPIO_InitStruct.GPIO_Pin = GPIO_Pin_12;
GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_Init(GPIOB, &GPIO_InitStruct);

/* Báo cho hệ thống biết chân PB12 là ngắt EXTI_Line12 */
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB, EXTI_PinSource12);

/* PB12 được kết nối tới EXTI_Line12 */
EXTI_InitStruct.EXTI_Line = EXTI_Line12;
/* Cho phép ngắt */
EXTI_InitStruct.EXTI_LineCmd = ENABLE;
/* Chọn chế độ là ngắt */
EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
/* Cho phép xảy ra ngắt theo cả sường tăng vào sườn giảm */
EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Rising_Falling;
/* Thêm vào EXTI */
EXTI_Init(&EXTI_InitStruct);

/* Thêm vector ngắt vào NVIC */
/* PB12 được kết nối tới EXTI_Line12, tương ứng với vector EXTI15_10_IRQn
*/
NVIC_InitStruct.NVIC_IRQChannel = EXTI15_10_IRQn;
/* Thiết lập độ ưu tiên */
NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0x00;
/* Thiết lập độ ưu tiên trong nhóm */
NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0x01;
```

```
/* Cho phép ngắt */
NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
/* Thêm vào NVIC */
NVIC_Init(&NVIC_InitStruct);
}

/* Viết chương trình con phục vụ ngắt */
/* Điều khiển ngắt tại chân PD0*/
void EXTI0_IRQHandler(void) {
        /* Kiểm tra trạng thái của cờ ngắt được thiết lập */
        if (EXTI_GetITStatus(EXTI_Line0) != RESET) {
        /* Các lệnh khi xảy ra ngắt */
        /* Xoá cờ ngắt */
        EXTI_ClearITPendingBit(EXTI_Line0);
        }}

/*Điều khiển ngắt tại chân PB12 */
void EXTI15_10_IRQHandler(void) {
        /*Kiểm tra xem cờ ngắt có được thiết lập không */
        if (EXTI_GetITStatus(EXTI_Line12) != RESET) {
        /* Các lệnh cần được thực hiện khi có ngắt xảy ra */
        /* Xoá cờ ngắt */
        EXTI_ClearITPendingBit(EXTI_Line12);
        }}
int main(void) {
        /* Khởi tạo hệ thống */
        SystemInit();
        /* Cấu hình ngắt PD0 */
        Configure_PD0();
```

```
    /* Cấu hình ngắt PB12 */
    Configure_PB12();
    while (1) {
}}
```

3.3. Lập trình Timer

3.3.1. Cơ bản về Timer trong STM32

Timer trong STM32 có rất nhiều chức năng chẳng hạn như bộ đếm counter, PWM, input capture, ngoài ra còn một số chức năng đặt biệt để điều khiển động cơ như encoder, hall sensors.

Timer là bộ định thời có thể sử dụng để tạo ra thời gian cơ bản dựa trên các thông số: clock, prescaler, autoreload, repetition counter.

Timer của STM32 là timer 16 bits có thể tạo ra các sự kiện trong khoảng thời gian từ nano giây tới vài phút gọi là UEV(update event).

3.3.2. Các thành phần cơ bản của một timer trong STM32

TIM_CLK: Xung Clock cung cấp cho timer.

PSC (prescaler): Là thanh ghi 16 bít làm bộ chia cho timer, có thể chia từ 1 tới 65535

ARR (auto-reload register): Là giá trị đếm của timer (16 bít hoặc 32 bít).

RCR (repetition counter register): Giá trị đếm lặp lại 16 bít.

Giá trị UEV được tính theo công thức sau:

$$\mathbf{UEV = TIM\_CLK/((PSC + 1)*(ARR + 1)*(RCR + 1))}$$

Ví dụ: TIM_CLK = 72MHz, PSC = 1, ARR = 65535, RCR = 0.

UEV = 72000000/((1 + 1)*(65535 + 1)*(1)) = 549.3 Hz

Với giá trị ARR (auto-reload) được cung cấp thì bộ định thời (timer) thực hiện các chế độ đếm khác nhau: đếm lên, đếm xuống hoặc kết hợp cả 2. Khi thực hiện đếm lên thì giá trị bộ đếm bắt đầu từ 0 và đếm tới giá trị ARR-1 thì báo tràn. Khi thực hiện đếm xuống thì bộ đếm bắt đầu từ giá trị ARR đếm xuống 1 thì báo tràn.

3.3.3. Các chế độ hoạt động của Timer

Timer có 4 chế độ cơ bản

- Input capture: Chân chụp đầu vào
- Output compare: So khớp trên chân ra
- PWM generation (Edge- and Center-aligned modes): Chế độ tạo xung
- One-pulse mode output: Đầu ra chế độ tạo một xung

3.3.4. Định nghĩa struct cấu hình cho timer cơ bản

typedef struct
{
uint16_t **TIM_Prescaler;** /*Giá trị tiền định được sử dụng để chia xung TIM; là một số nằm trong khoảng từ 0x0000 đến 0xFFFF */
uint16_t **TIM_CounterMode**; /*Chỉ định chế độ đếm; có thể là một giá trị của @ref TIM_Counter_Mode */
uint32_t **TIM_Period**; /* Chỉ định giá trị sẽ được tải vào thanh ghi Auto-Reload Register tại sự kiện update tiếp theo và phải là giá trị nằm trong khoảng 0x0000 đến 0xFFFF */
uint16_t **TIM_ClockDivision**; /*Chỉ địnhg giá trị chia Clock; có thể được sử dụng bởi @ref TIM_Clock_Division_CKD */
uint8_t **TIM_RepetitionCounter;** /*!< Chỉ định giá trị lặp lại của Counter value. Mỗi lần RCR đếm lùi và đạt tới 0, một sự kiện update sẽ được sinh ra và khởi động bộ đếm từ RCR (N). Trong chế độ PWM (N+1) yêu cầu:

+) Số chu kỳ PWM trong chế độ edge-aligned
+) Số chu kỳ PWM trong chế độ center-aligned

Tham số này phải là giá trị nằm trong khoảng 0x00 đến 0xFF.
@chú ý: Tham số này chỉ hợp lệ cho các timer từ TIM1 đến TIM8. */
} **TIM_TimeBaseInitTypeDef**;

- TIM_Prescaler: Tham số TIM_Prescaler hiểu đơn giản như một bộ chia tần số. Thí dụ STM32F4 Tần số cao nhất mà clock timer 4 đạt được là 84Mhz sau khi qua bộ chia này sẽ ra tần số clock timer(Fc_timer). Nếu chọn Fc_timer = 1Mhz từ Fc_timer = 84000000/84. TIM_Prescaler có công thức là:

((SystemCoreClock/2)/1000000)-1 = 83 ,

do hệ đếm bắt đầu từ 0 chứ không phải là 1 nên bắt đầu đếm từ 0 -> 83 sẽ, tức là 84 giá trị.

TIM_Prescaler = ((SystemCoreClock/n)/Fc_timer)-1

Chú ý: Tùy vào timer nào mà chỉ số chia n sẽ khác nhau. Ví dụ trong STM32F4 gồm có những timer và hệ số chia khác nhau như hình bên dưới :

TIMER	TYPE	RESOLUTION	PRESCALER	CHANNELS	MAX INTERFACE CLOCK	MAX TIMER CLOCK*	APB
TIM1, TIM8	Advanced	16bit	16bit	4	SysClk/2	SysClk	2
TIM2, TIM5	General purpose	32bit	16bit	4	SysClk/4	SysClk, SysClk/2	1
TIM3, TIM4	General purpose	16bit	16bit	4	SysClk/4	SysClk, SysClk/2	1
TIM9	General purpose	16bit	16bit	2	SysClk/2	SysClk	2
TIM10, TIM11	General purpose	16bit	16bit	1	SysClk/2	SysClk	2
TIM12	General purpose	16bit	16bit	2	SysClk/4	SysClk, SysClk/2	1
TIM13, TIM14	General purpose	16bit	16bit	1	SysClk/4	SysClk, SysClk/2	1
TIM6, TIM7	Basic	16bit	16bit	0	SysClk/4	SysClk, SysClk/2	1

Bảng 3.4 Timer và các hệ số chia của STM32F4

- **TIM_CounterMode**: Thiết lập mode cho timer là đếm lên hay đếm xuống. Nếu chọn mode đếm tăng có nghĩa là mỗi xung nhịp timer, bộ đếm counter sẽ tự tăng lên một giá trị theo chiều dương cho đến khi nào bằng giá trị period sẽ đếm lại từ đầu, người ta thường gọi trường hợp này là tràn bộ đếm.

- **TIM_Period**: Period có nghĩa là chu kỳ của timer (không phải là chu kỳ của 1 xung clock timer). Ví dụ một chu kỳ gồm 1000 xung clock mà mỗi xung clock = 1us ta sẽ được period là 1ms.

Chú ý: Khi cấu hình sử dụng timer ta cần quan tâm đến 3 yếu tố chính đó là :

+) Đếm với xung clock timer là bao nhiêu (Fc_timer – xác định qua TIM_Prescaler).

+) Đếm lên hay đếm xuống (TIM_CounterMode).

+) Đếm đến bao nhiêu (TIM_Period).

3.3.5. Thí dụ

Thí dụ 1: Cấu hình cho timer 3 trong STM32F1 tạo ngắt với 100ms

Phân tích**: (lưu ý 1 giây = 1ms x 1000)**.

- Tần số cao nhất mà timer 3 trong STM32F1 đạt được là 72Mhz. Yêu cầu cần timer 3 tạo interrupt mỗi 100ms.

- Giả sử cần Counter của timer đếm 100 lần để được 100ms và sinh ngắt thì ta có: Tần sô Fc_timer3 = (100ms x 100 clock) x 10 = 10.000 clock/ 1 giây (10Khz).

Kết quả:

+) TIM_Prescaler = (72000000/10.000 – 1); //Fc_timer là 10khz

+) TIM_Period = 100 lần – 1 = 99

- Giả sử cần Counter timer đếm 1000 lần để được 100ms và sinh ngắt thì tương tự: Tần số Fc_timer3 = (100ms x 1000 clock) x 10 = 100.000 clock/1 giây (100Khz)

Kết quả:

+) TIM_Prescaler = (72000000/100.000 – 1); //Fc_timer là 100khz

+) TIM_Period = 1000 lần – 1 = 999

Cấu hình các thông số cho timer3 với trường hợp period = 99:

```
TIM_TimeBaseStructure.TIM_Period = 99; // delay 10ms
TIM_TimeBaseStructure.TIM_Prescaler = (72000000/10000 – 1); //10khz
TIM_TimeBaseStructure.TIM_ClockDivision = 0;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
```

Trong Thí dụ này, chọn Fc_timer3 = 10000 (10khz) nghĩa là :

-> Tần số timer 10.000 xung/1giây tương ứng 10 xung/1ms nên tần số cho 10ms là 100xung/10ms

Thí dụ 2: Ví dụ cấu hình cho timer 4 trong STM32F4 sinh ngắt sau mỗi 1ms

Phân tích: Tần số cao nhất mà mà timer 4 trong STM32F4 đạt được là 84Mhz - Yêu cầu cần timer 4 tạo interrupt mỗi 1ms: Giả sử cần counter của timer đếm 100 lần để được 1ms và sinh ngắt thì ta có: Tần sô Fc_timer4 = (1ms x 100 clock) x 1000 = 100.000 clock/ 1 giây (100Khz).

Kết quả:

+) TIM_Prescaler = (84000000/100.000 – 1); //Fc_timer là 100khz

+) TIM_Period = 100 lần – 1 = 99

- Giả sử cần counter timer đếm 1000 lần để được 1ms và sinh ngắt thì tương tự: Tần số Fc_timer4 = (1ms x 1000 clock) x 1000 = 1000.000 clock/1giây (1Mhz).

Kết quả:

+) TIM_Prescaler = (72000000/1000.000 – 1); //Fc_timer là 1Mhz

+) TIM_Period = 1000 lần – 1 = 999 ;

Cấu hình các thông số cho timer4 với trường hợp period = 999:

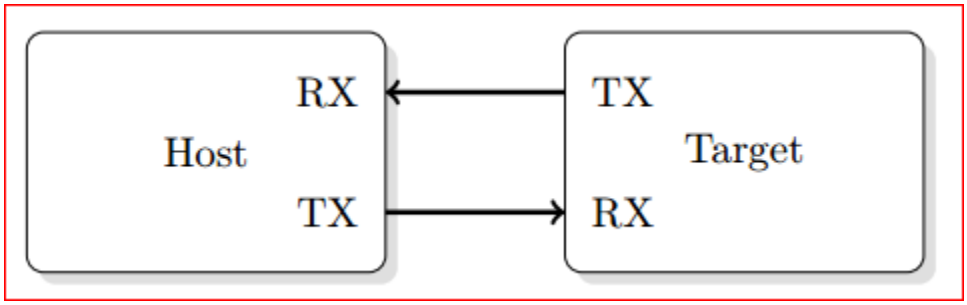
```
// f = 1Mhz
TIM_TimeBaseStructure.TIM_Prescaler = ((SystemCoreClock/2)/1000000)-1;
TIM_TimeBaseStructure.TIM_Period = 1000 – 1;
TIM_TimeBaseStructure.TIM_ClockDivision = 0;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure);
```

Trường hợp timer 4 trong STM32F4, period = 999; Yêu cầu đếm 1000 xung để được **ngắt 1ms** thì 1 xung sẽ tương ứng với 1us; Hệ số chia TIM_Prescaler = 1.000.000 trong khi xung hệ thống cấp là 84.000.000 do đó mỗi xung của clock timer sẽ bằng 84 xung của system clock.

3.4. Lập trình truyền thông qua UART

3.4.1. Giới thiệu chuẩn truyền thông nối tiêp

Phương pháp cơ bản nhất để giao tiếp với một bộ xử lý nhúng là nối tiếp không đồng bộ. Truyền thông nối tiếp không đồng bộ, thông tin được truyền trên 2 đường dây đối xứng được kết nối với nhau. Ở đây sẽ đề cập đến host và target.

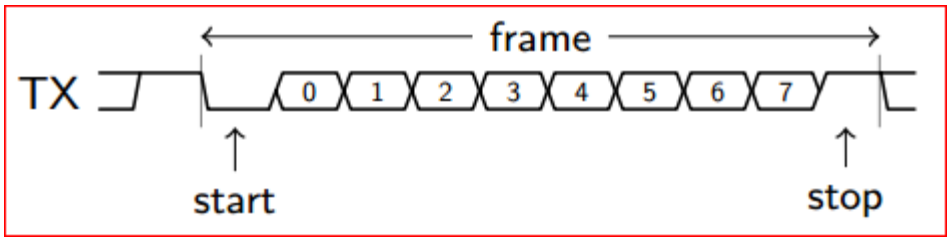


Hình 3.2. 2 chân truyền và nhận dữ liệu cơ bản trên STM32

Khi dữ liệu gửi từ Host sang Target, nó gửi một chuỗi bit mã hóa trên đường dây truyền TX của nó, Target nhận dữ liệu trên đường dây nhận RX của nó. Tương tự, Target gửi dữ liệu sang Host, nó gửi một chuỗi bit mã hóa trên đường dây truyên TX của nó, Host nhận dữ liệu trên đường dây nhận RX của nó. Chế độ này được gọi là truyền thông không đồng bộ bởi vì host và target có thời gian không đồng bộ với nhau. Thay vào đó, chuỗi bit được mã hóa ở bên phát và giải mã ở bên nhận.

Một thiết bị thường được sử dụng để mã hóa và giải mã chuỗi bit không đồng bộ như vậy là UART, nó chuyển đổi các byte dữ liệu bởi phần mềm vào một chuỗi các bit riêng lẻ và ngược lại, chuyển đổi một chuỗi các bit vào byte dữ liệu được truyền tới phần mềm. Vi điều khiển STM32 có 5 thiết bị được gọi là USART(universal synchronous/ asynchronous receiver/ transmitter) bởi vì nó có chế độ truyền thông không đồng nhất với nhau. Trong chương này, chúng ta sẽ truyền thông nối tiếp giữa Target là STM32 USART và host là bộ chuyển đổi USB/UART.

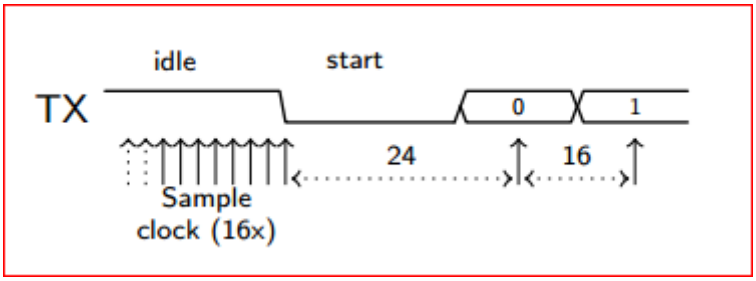
UARTs cũng có thể được sử dụng để giao tiếp với các thiết bị ngoại vi khác. Ví dụ như Modem di động GSM/GPRS hay Modem Blutooth được sử dụng rộng rãi với giao diện bộ điều khiển UART. Như các giao diện khác, trong tương lai giao thức truyền thông nối tiếp sẽ tiếp cận được với nhiều loại thiết bị khác.



Hình 3.3. Giao thức truyền thông nối tiếp

Một trong những cách giải mã cơ bản được sử dụng cho truyền thông nối tiếp không đồng bộ được minh họa như hình trên. Mỗi ký tự được truyền trong một khung, bắt đầu là 1 bit thấp sau đó là 8 bit dữ liệu và cuối cùng là 1 bit cao(bit dừng). Dữ liệu được mã hóa thành các mức tín hiệu cao và thấp (1 và 0). Giữa các khung, tình trạng nhàn rỗi được báo hiệu bằng cách truyền một tín hiệu cao liên tục. Mỗi khung được đảm bảo để bắt đầu với một sự chuyển tiếp high-low và chứa ít nhất một quá trình chuyển đổi từ mức cao xuống mức thấp. Lựa chọn cấu trúc khung cơ bản bao gồm số các bít dữ liệu khác nhau, một bít chẵn lẻ sau bit dữ liệu cuối cùng để phát hiện lỗi và điều kiện để dừng.

Không có tín hiệu clock được mã hóa trực tiếp trong tín hiệu (ngược lại với giao thức báo hiệu khác như mã hóa Manchester), quá trình chuyển đổi chỉ cung cấp thông tin theo thời gian trong dòng dữ liệu. Ở bên phát và bên thu, mỗi bên duy trì một đồng hồ chạy độc lập (là bội số của) tần số chung và không chính xác, được gọi là tốc độ Baud. Hai đồng hồ chưa được đồng bộ và không đảm bảo độ chính xác ở cùng một tần số, nhưng 2 tần số phải gần giống nhau để có thể khôi phục được dữ liệu.



Hình 3.4. Giải mã tín hiệu UART.

Để hiểu làm thế nào bên nhận hiểu được tín hiệu mã hóa, giả sử có 1 đồng hồ chạy với tốc độ Baud(16x), bắt đầu từ trạng thái nhàn rỗi (idle) như hình trên.

Bên nhận nhận mẫu tín hiệu RX cho đến khi nó phát hiện một quá trình chuyển đổi cao thấp, sau đó nó chờ trong thời gian 1,5 bit (24 chu kỳ đồng hồ) để lấy mẫu tín hiệu RX của nó và ước tính bit 0 là dữ liệu trung tâm. Bên nhận nhận mẫu tín hiệu RX sau 16 chu kỳ đồng hồ, khi đã đọc 7 bit dữ liệu và cả bit dừng. Từ thời điểm đó, quá trình được lặp đi lặp lại. Dữ liệu đã truyền thành công trên một khung yêu cầu, hơn 10,5 chu kỳ bit, sự chênh lệch xung clock ở bên thu và bên phát và 0,5 bit là dành cho giai đoạn phát hiện bit dừng.

3.4.2. Giao thức truyền thông UART trên STM32

Hình thức truyền thông đơn giản nhất là dựa trên trạng thái của thiết bị USART. Mỗi STM32 USART có 6 thanh ghi – 4 thanh ghi được sử dụng để cấu hình phần cứng thông qua việc khởi tạo trong thư viện. Hai thanh ghi còn lại là thành ghi dữ liệu và thanh ghi điều khiển. Thanh ghi dữ liệu chiếm 1 vị trí bộ nhớ duy nhất, nó thực sự là 2 vị trí riêng biệt, khi thanh ghi dữ liệu được ghi vào thì dữ liệu được truyền bởi USART, khi thanh ghi dữ liệu đọc, các ký tự gần nhất sẽ nhận được do USART gửi trả lại. Thanh ghi trạng thái chứa 1 số cờ để xác định trạng thái của USART. Quan trọng nhất trong số đó là:

USART_FLAG_TXE -- Transmit data register empty (Cờ báo đệm truyền dữ liệu rỗng)

USART_FLAG_RXNE -- Receive data register not empty (Cờ đệm nhận dữ liệu rỗng)

Các cờ khác cung cấp thông tin như kiểm tra lỗi chẵn lẻ, lỗi khung và tràn bít, cần được kiểm tra trong việc thực thi.

Để truyền 1 ký tự phần mềm ứng dụng phải đợi cho đến khi thanh ghi dữ liệu rỗng, sau đó mới ghi dữ liệu được vào thanh ghi. Trạng thái của thanh ghi dữ liệu được xác định bởi cờ USART_FLAG_TXE của thanh ghi trạng thái USART. Đoạn code sau thực hiện việc ghi dữ liệu vào đệm dữ liệu cơ bản trong USART1.

```
int putchar( int c){
while ( USART_GetFlagStatus (USART1 , USART_FLAG_TXE ) == RESET);
USART1 ->DR = (c & 0xff);
```

return 0;

}

Với 1 thanh ghi dữ liệu truyền đơn lẻ, nó sẽ thực hiện nhanh như tốc độ Baud và phải đợi cờ trạng thái giữa các ký tự.

Đoạn code sau thực hiện việc đọc một ký tự cơ bản:

int getchar(void){

while (USART_GetFlagStatus (USART1 , USART_FLAG_RXNE) == RESET);

return USART1 ->DR & 0xff;

}

Cờ trạng thái USART_FLAG_RXNE được sử dụng khi muốn nhận một ký tự.

3.2.3. Khởi tạo USART

Giống như tất cả các ngoại vi của STM32, USART phải được khởi tạo trước khi được sử dụng. Việc khởi tạo bao gồm cấu hình chân, bật xung clock và khởi tạo thiết bị ngoại vi. Đối với STM32F10X thì có 3 USART lần lượt là USART1, USART2 và USART3.

Có 3 module mà mỗi phần của thư viện được yêu cầu cho ứng dụng của USART(bao gồm các tập tin liên quan đến đối tượng của bạn).

include <stm32f10x .h>

include <stm32f10x_gpio .h>

include <stm32f10x_rcc .h>

include <stm32f10x_usart .h>

Bước đầu tiên cho việc khởi tạo là cho phép tín hiệu RCC (thiết lập và khởi động lại xung clock) cho các khối chức năng khác nhau cần thiết cho việc sử dụng USART bao gồm các cổng GPIO (như cổng A cho USART1), các thành phần của USART và module thay thế AF khác. Với USART, các bước khởi tạo với RCC như sau:

RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1

|RCC_APB2Periph_AFIO |

RCC_APB2Periph_GPIOA , ENABLE);

Với USART2 là thiết bị ngoại vi APB1, vì thế khởi tạo RCC sẽ hơi khác một chút. Thông báo cho cờ APB2 để nó cho phép xung clock được hoạt động trong mỗi bước.

Sau khi bật xung clock, tiếp theo là cấu hình cho các chân tương ứng với 3 USART, mặc định như trong bảng dưới đây:

Function	Pin		
	x=1	x=2	x=3
USARTx_TX	PA9	PA2	PB10
USARTx_RX	PA10	PA3	PB11
USARTx_CK	PA8	PA4	PB12
USARTx_RTS	PA12	PA1	PB14
USARTx_CTS	PA11	PA0	PB13

Hình 3.4. Các chân UART trên kít STM32F4

Trong tài liệu hướng dẫn, STM32 đã cung cấp những thông tin cho việc cấu hình các chân GPIO tương ứng với các thiết bị ngoại vi khác.

Đối với các USARTs, thông tin được thể hiện trong bảng dưới đây:

USART pinout	Configuration	GPIO Configuration
USARTx_TX	Full Duplex	Alternate function push-pull
	Half duplex Synchronous mode	Alternate function push-pull
USARTx_RX	Full Duplex	Input floating/Input Pull-up
	Half duplex Synchronous mode	Not used. Can be used as General IO
USARTx_CK	Synchronous mode	Alternate function push-pull
USARTx_RTS	Hardware flow control	Alternate function push-pull
USARTx_CTS	Hardware flow control	Input floating/Input pull-up

Hình 3.5. Các chế độ chân của UART

Như đã đề cập trước đó, các USARTs trong STM32 có khả năng hỗ trợ một chế độ hoạt động khác - đồng bộ nối tiếp mà đòi hỏi một tín hiệu đồng hồ riêng biệt (USARTx_CK) mà chúng ta sẽ không được sử dụng. Ngoài ra, các USART còn có khả năng hỗ trợ " điều khiển phần cứng" (tín hiệu USARTx_RTS và

USARTx_CTS). Để truyền thông nối tiếp cơ bản ta phải cấu hình cho 2 chân USART1_Tx và USART1_Rx. Đầu ra được định hướng bởi các thành phần USART, khi nó là đầu vào thì có thể được cấu hình ở chế độ floating or pulled up. Các chức năng và hằng số để cấu hình cho chân được định nghĩa trong thư viện stm32f10x_gpio.h.

Đoạn code sau minh họa cho việc cấu hình chân cho USART1_Tx và USART1_Rx.

```
GPIO_InitTypeDef   GPIO_InitStruct ;
GPIO_StructInit (&GPIO_InitStruct );
// Initialize USART1_Tx
GPIO_InitStruct.GPIO_PIN = GPIO_Pin_9 ;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz ;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP ;
GPIO_Init (GPIOA,& GPIO_InitStruct );
// Initialize USART1_RX
GPIO_InitStruct.GPIO_PIN = GPIO_Pin_10 ;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN_FLOATING ;
GPIO_Init (GPIOA,& GPIO_InitStruct );
```

Bước khởi tạo cuối cùng là khởi tạo cho USART, chúng ta sử dụng khởi tạo mặc định cho USART với tốc độ Baud là 9600, 8 bit dữ liệu, 1 bit stop, không có bit kiểm tra chẵn lẻ và không có điều khiển bởi các thủ tục của thư viện USART_StructInit.

Để thay đổi các khởi tạo được mặc định, ta thay đổi các thông số của các trường tương ứng trong USART_InitStructure.

```
USART_InitTypeDef USART_InitStructure ;
// Initialize USART
USART_StructInit (&USART_InitStructure );
USART_InitStructure.USART_BaudRate = 9600;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx ;
```

USART_Init (USART1 ,& USART_InitStructure);

USART_Cmd (USART1 , ENABLE);

Thí dụ: Chương trình truyền chữ hello world từ kít STM32 lên máy tính.

Các đoạn code trong các phần trước cung cấp tất cả các chức năng cơ bản cần thiết để truyền thành công một USART STM32. Dưới đây sẽ là 1 tiến trình tạo ra một ứng dụng đơn giản là gửi chữ " hello world" từ STM32 về máy tính. Đây là ứng dụng đầu tiên đòi hỏi phải kết nối các thiết bị phần cứng với nhau. Vì vậy cần có thiết bị chuyển đổi USB to COM.

Các hàm chức năng như sau:

Mở cổng USART:

int uart_open (USART_TypeDef * USARTx , uint32_t baud , uint32_t flags);

Đóng cổng sau khi kết thúc truyền thông:

int uart_close (USART_TypeDef * USARTx);

Gửi 1 ký tự qua cổng

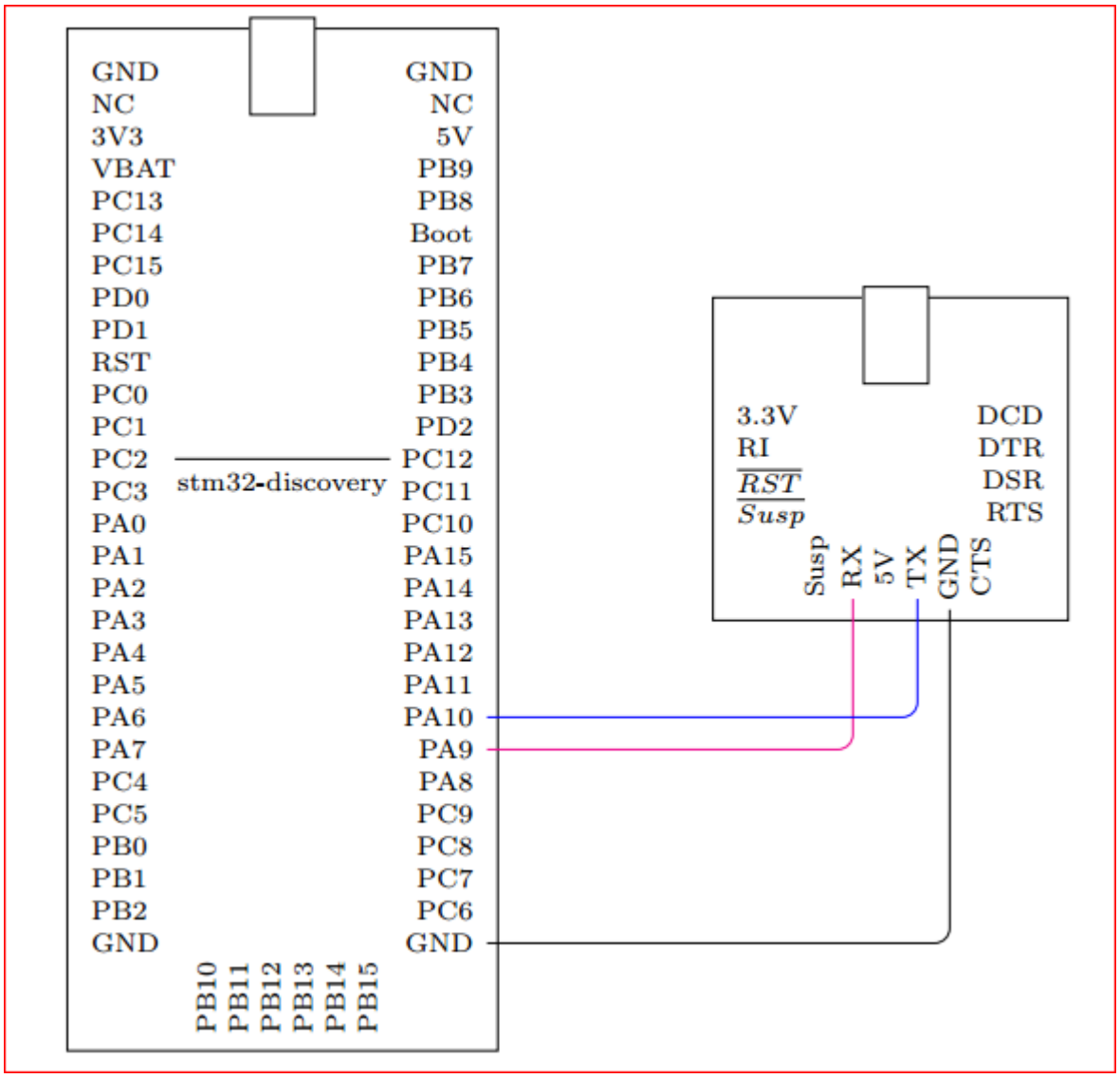
int uart_putc (int c, USART_TypeDef * USARTx);

Nhận một ký tự từ cổng:

int uart_getc (USART_TypeDef * USARTx);

Chức năng của hàm uart_open là:

1. Thiết lập xung clock cho gpio/usart
2. Cấu hình cho chân của usart
3. Cấu hình và cho phép usart1 hoạt động



Hình 3.6: Sơ đồ nối dây với USART1

Trong file usart bao gồm:

include <stm32f10x .h>

include <stm32f10x_rcc .h>

include <stm32f10x_gpio .h>

include <stm32f10x_usart .h>

include " uart.h "

Chức năng của hàm uart_getchar và uart_putchar dùng để đọc và ghi từng ký tự. Hai hàm này được sử dụng giống như trong Linux. Trong hàm main.c bao gồm:

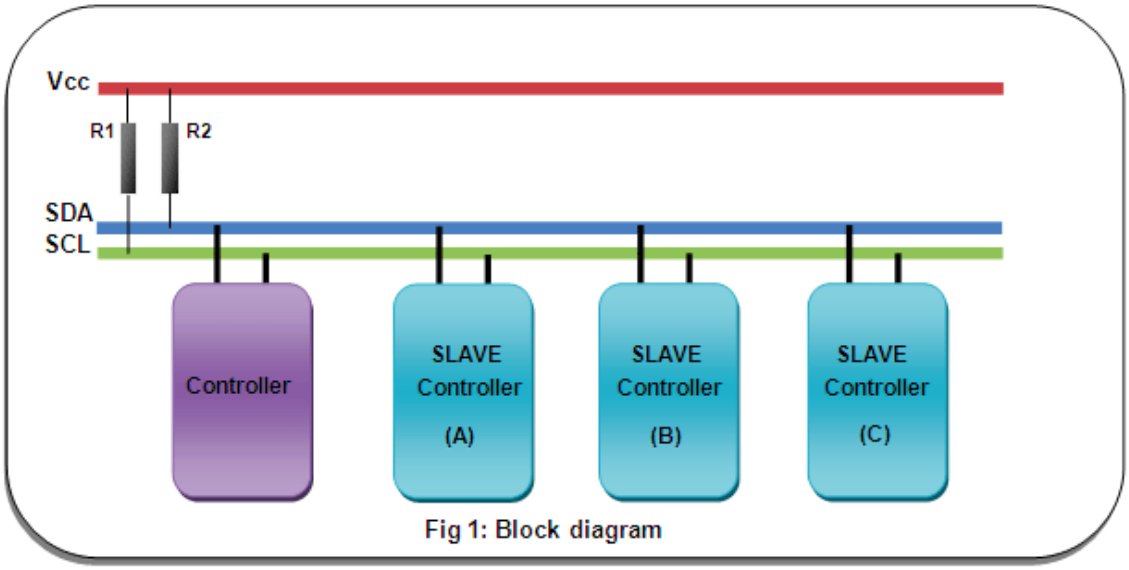
1. Thiết lập thời gian tick
2. Thiết lập cho usart
3. Viết dòng chữ " Hello Word với thời gian trễ là 25ms".

Trong file main.c phải bao gồm các thư viện: stm32f10x.h, stm32f10x_usart.h và uart.h. Chúng ta chỉ cần 3 dây để kết nối như minh họa ở hình trên. Tín hiệu GND ở Board mạch và Bộ chuyển đổi uart được nối với nhau. Có nhiều chân GND trên Board mạch, tất cả đều được kết nối điện bởi PCB. Do đó, chỉ cần 1 dây duy nhất (thường là dây màu đen) để nối các tín hiệu đất chung với nhau. Sau đó, chúng ta cần nối các tín hiệu tx(rx) của USART1 tương ứng với chân rx (tx) của bộ chuyển đổi USB/uart.

3.5. Lập trình qua giao diện I2C

3.5.1. Giới thiệu truyền thông I2C

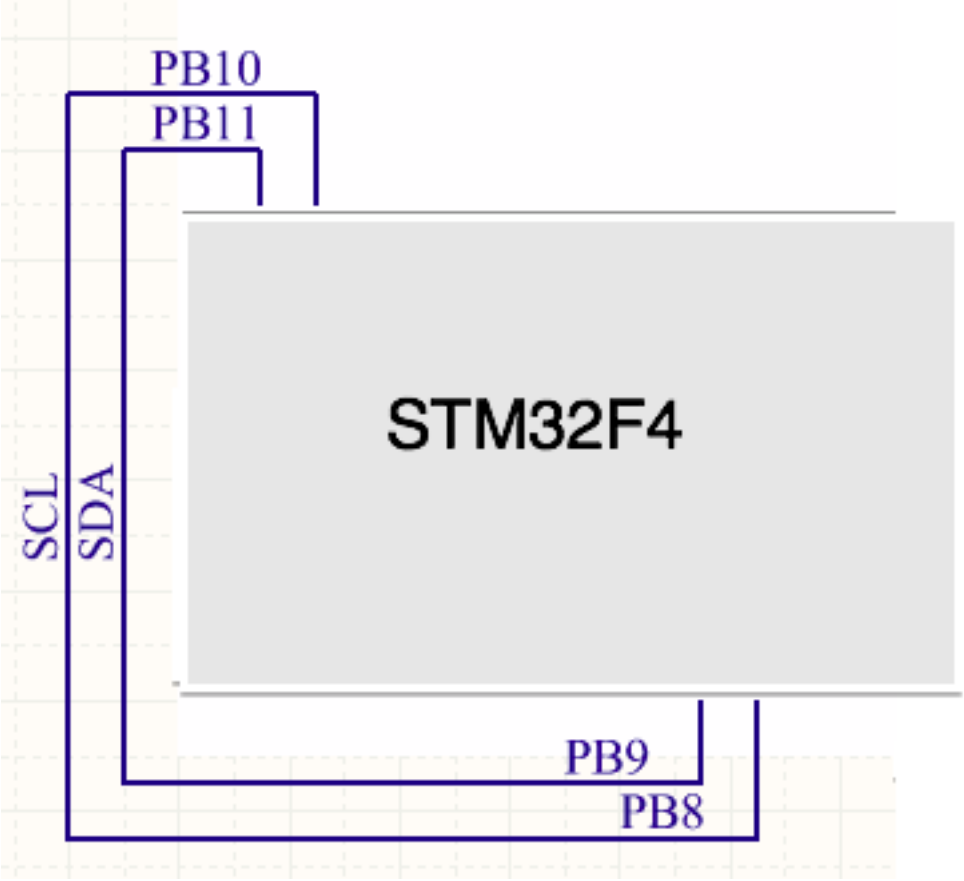
I2C (inter-integrated circuit) là chuẩn truyền thông nối tiếp 2 dây gồm 1 dây xung clock(SCL) và 1 dây dữ liệu (SDA). Các chip chủ-tớ được nối chung với nhau trên hai đường dây này và được nối với điện trở treo.



Hình 3.7. Sơ đồ khối giao diện SPI

3.5.2. Kết nối phần cứng

Sơ đồ kết nối phần cứng như bên dưới:



Hình 3.8. Thí dụ sơ đồ kết nối phần cứng qua I2C

3.5.3. Lập trình I2C cơ bản

Trong thí dụ này sử dụng 2 bộ I2C, I2C1 vai trò là chip master, I2C2 là chip slave. Master truyền dữ liệu theo kiểu thông thường không thiết lập ngắt và DMA, riêng slave thiết lập ngắt sự kiện.

Giải thích hàm I2C_Configuration() từ dòng 35 ->102:

Dòng 37 –> 43: Đây là các đoạn mã tiền xử lý trong C, điển hình ở dòng 37 #ifdef FAST_I2C_MODE có nghĩa là nếu như macro FAST_I2C_MODE được định nghĩa thì sẽ thực hiện 2 dòng 38-39, nếu FAST_I2C_MODE chưa được định nghĩa thì sẽ thực hiện 2 dòng 40-41. Ở dòng 8 đã định nghĩa sử dụng FAST_I2C_MODE.

Dòng 51: Để sử dụng I2C thì phải chọn mode OD (open drain) cấu hình cho GPIO.

Dòng 52: Do kết nối phần cứng không sử dụng điện trở treo lên mức cao nên

phải cấu hình điện trở treo nội lên mức cao.

Dòng 63: Lệnh này dùng reset lại các thanh ghi của khối I2C đưa về mặc định.

Dòng 64: Chọn mode hoạt động là I2C, ngoài ra còn một số chuẩn khác dựa trên chuẩn I2C như SMBus.

Dòng 65: Chọn tỉ lệ mức thấp/mức cao của xung clock. Macro I2C_DUTYCYCLE này đã được định nghĩa phía bên trên ở dòng 37.

Dòng 66: Địa chỉ của thiết bị I2C, ở chip master có thể không cần thiết do chip master là chip chủ động, chip muốn được giao tiếp với chip khác.

Dòng 67: Bật xác nhận ACK.

Dòng 68: Thiết lập tốc độ xung clock trên chân SCL, macro đã được định nghĩa ở dòng 37.

Dòng 69: Chọn chế độ địa chỉ 7 bit, mạng I2C sẽ có tối đa là 128 thiết bị.

Phần thiết lập cho slave tương tự như master, ở dòng 81 thiết lập địa chỉ của thiết bị slave, điều này là rất quan trọng, trong giao tiếp I2C để master giao tiếp với slave thì master phải biết được địa chỉ của thiết bị slave đó, thí dụ này định nghĩa địa chỉ này là 0x68. Dòng 89 thiết lập ngắt lỗi và ngắt sự kiện trên slave, mỗi khi có một hoạt động gì đó trên đường truyền thì slave sẽ phân tích các sự kiện này ở hàm ngắt.

Kết thúc hàm I2C_Configuration():

Dòng 12: Tạo mảng MasterTxBuffer[] để chứa dữ liệu mà master truyền đi, và khởi tạo sẵn dữ liệu bên trong mảng này là 1, 2, 3.

Dòng 13: Mảng chứa dữ liệu mà master sẽ nhận về.

Dòng 14: Mảng chứa dữ liệu slave phát đi.

Dòng 15: Mảng chứa dữ liệu mà slave nhận về. Mảng này sẽ được hàm ngắt cập nhật dữ liệu.

Ở chương trình chính tôi sử dụng hàm I2C_ByteWrite() để phát dữ liệu của mảng MasterTxBuffer[] đến slave, và hàm I2C_BufferRead() để đọc dữ liệu từ slave phát đến master, dữ liệu nhận về sẽ được gán vào mảng MasterRxBuffer[].

Hai hàm này được viết trong thư viện MY_I2C.h.

Giải thích file stm32f4xx_it.c. Quá trình slave xử lý chủ yếu nằm trong file này:

Ở dòng 74: Mỗi khi có một sự kiện truyền nhận dữ liệu, địa chỉ,… của slave sẽ xảy ra một ngắt và sẽ được trỏ đến hàm này.

Giả sử xét trường hợp slave truyền dữ liệu, để hoàn tất quá trình này sẽ sinh ra rất nhiều ngắt, các bước mà ngắt lần lược thực hiện:

Bước 1: Khi phát hiện ra một sự kiện trên đường truyền đầu tiên nó sẽ nhảy vào ngắt I2C_EVENT_SLAVE_TRANSMITTER_ADDRESS_MATCHED lợi dụng việc này tôi gán Tx_Idx = 0 biến này là một biến đếm thứ tự dữ liệu gửi đi.

Bước 2: Nếu master muốn slave gửi dữ liệu thì lần ngắt tiếp theo sẽ nhảy vào đây, ở ngắt này sẽ gửi dữ liệu trong mảng SlaveTxBuffer[] đến master. Ngắt này sẽ được thực hiện nhiều lần, chương trình thực hiện phép tăng biến Tx_Idx mỗi lần xảy ra ngắt. Quá trình xảy ra cho đến khi master không muốn nhận dữ liệu nữa nó sẽ phát ra một tín hiệu kết thúc.

Bước 3: Bước cuối trong quá trình truyền dữ liệu của slave, nó sẽ nhảy đến ngắt I2C_EVENT_SLAVE_STOP_DETECTED ở dòng 113.

Quá trình nhận dữ liệu của slave cũng xảy ra tương tự như 3 bước trên nhưng các sự kiện có khác để phù hợp với quá trình nhận với các dòng từ 101 -> 106.

File **main.c:**

```
001  #include "stm32f4xx.h"
002  #include "MY_I2C.h"
003
004  /*  MASTER                SLAVE
005    PB8 --- I2C1_SCL                  PB10 --- I2C2_SCL
006    PB9 --- I2C1_SDA                  PB11 --- I2C2_SDA
007  */
008  #define FAST_I2C_MODE
009  #define Slave_Address 0x68
010  #define BufferSIZE 3
011
```

```
012  volatile uint8_t MasterTxBuffer[BufferSIZE] = {1 , 2 , 3};
013  volatile uint8_t MasterRxBuffer[BufferSIZE];
014  volatile uint8_t SlaveTxBuffer[BufferSIZE] = {4 , 5 , 6};
015  volatile uint8_t SlaveRxBuffer[BufferSIZE];
016
017  GPIO_InitTypeDef  GPIO_InitStructure;
018  I2C_InitTypeDef   I2C_InitStructure;
019  NVIC_InitTypeDef  NVIC_InitStructure;
020
021  void I2C_Configuration(void);
022  void Delay(__IO uint32_t nCount);
023
024  int main(void)
025  {
026    I2C_Configuration();
027    while (1)
028    {
029      I2C_ByteWrite(I2C1, Slave_Address, (u8*)MasterTxBuffer,3);
030      I2C_BufferRead(I2C1, Slave_Address, (u8*)MasterRxBuffer,3);
031      while(1);
032    }
033  }
034
035  void I2C_Configuration(void)
036  {
037  #ifdef FAST_I2C_MODE
038   #define I2C_SPEED 400000
039   #define I2C_DUTYCYCLE I2C_DutyCycle_16_9
040  #else /* STANDARD_I2C_MODE*/
041   #define I2C_SPEED 100000
042   #define I2C_DUTYCYCLE I2C_DutyCycle_2
043  #endif /* FAST_I2C_MODE*/
044
045    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
```

```
046    RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);
047    RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C2, ENABLE);
048
049 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11;
050    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
051    GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;
052    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;  // enable pull up resistors
053    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
054    GPIO_Init(GPIOB, &GPIO_InitStructure);
055
056    GPIO_PinAFConfig(GPIOB,GPIO_PinSource8,GPIO_AF_I2C1);   // only connect to
057    GPIO_PinAFConfig(GPIOB,GPIO_PinSource9,GPIO_AF_I2C1);   // only connect to
058    GPIO_PinAFConfig(GPIOB,GPIO_PinSource10,GPIO_AF_I2C2);   // only connect to
059    GPIO_PinAFConfig(GPIOB,GPIO_PinSource11,GPIO_AF_I2C2);   // only connect to
060
061/*****************************Master ******************/
062    /* I2C De-initialize */
063    I2C_DeInit(I2C1);
064    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
065    I2C_InitStructure.I2C_DutyCycle = I2C_DUTYCYCLE;
066    I2C_InitStructure.I2C_OwnAddress1 = 0x01;
067    I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
068    I2C_InitStructure.I2C_ClockSpeed = I2C_SPEED;
069       I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
070    I2C_Init(I2C1, &I2C_InitStructure);
071   /* I2C ENABLE */
072    I2C_Cmd(I2C1, ENABLE);
073    /* Enable Interrupt */
074    //I2C_ITConfig(I2C1, (I2C_IT_ERR ) , ENABLE);
075    //I2C_ITConfig(I2C1, (I2C_IT_ERR | I2C_IT_EVT | I2C_IT_BUF) , ENABLE);
076   /************************************* Slave ******************************/
077    /* I2C De-initialize */
078    I2C_DeInit(I2C2);
079    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
```

```
080   I2C_InitStructure.I2C_DutyCycle = I2C_DUTYCYCLE;
081   I2C_InitStructure.I2C_OwnAddress1 = Slave_Address;
082   I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
083   I2C_InitStructure.I2C_ClockSpeed = I2C_SPEED;
084   I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
085   I2C_Init(I2C2, &I2C_InitStructure);
086   /* I2C ENABLE */
087   I2C_Cmd(I2C2, ENABLE);
088   /* Enable Interrupt */
089   I2C_ITConfig(I2C2, (I2C_IT_ERR | I2C_IT_EVT | I2C_IT_BUF) , ENABLE);
090
091   NVIC_InitStructure.NVIC_IRQChannel = I2C2_EV_IRQn;
092   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
093   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
094   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
095   NVIC_Init(&NVIC_InitStructure);
096
097   NVIC_InitStructure.NVIC_IRQChannel = I2C2_ER_IRQn;
098   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
099   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
100   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
101   NVIC_Init(&NVIC_InitStructure);
102 }
103
104 void Delay(__IO uint32_t nCount)
105 {
106   while(nCount--)
107   {
108   }
109 }
110
111 #ifdef  USE_FULL_ASSERT
112 void assert_failed(uint8_t* file, uint32_t line)
113 {
```

```
114   /* User can add his own implementation to report the file name and line number,
115      ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
116
117   /* Infinite loop */
118   while (1)
119   {
120   }
121 }
122 #endif
```

File **stm32f4xx_it.c**:

```
001 #include "stm32f4xx_it.h"
002
003 void NMI_Handler(void)
004 {
005 }
006
007 void HardFault_Handler(void)
008 {
009   /* Go to infinite loop when Hard Fault exception occurs */
010   while (1)
011   {
012   }
013 }
014
015 void MemManage_Handler(void)
016 {
017   /* Go to infinite loop when Memory Manage exception occurs */
018   while (1)
019   {
020   }
021 }
022
023 void BusFault_Handler(void)
024 {
```

```
025   /* Go to infinite loop when Bus Fault exception occurs */
026   while (1)
027   {
028   }
029 }
030
031 void UsageFault_Handler(void)
032 {
033   /* Go to infinite loop when Usage Fault exception occurs */
034   while (1)
035   {
036   }
037 }
038
039 void SVC_Handler(void)
040 {
041 }
042
043 void DebugMon_Handler(void)
044 {
045 }
046
047 void PendSV_Handler(void)
048 {
049 }
050
051 void SysTick_Handler(void)
052 {
053 }
054
055 void I2C2_ER_IRQHandler(void)
056 {
057   /* Check on I2C2 AF flag and clear it */
058   if (I2C_GetITStatus(I2C2, I2C_IT_AF))
```

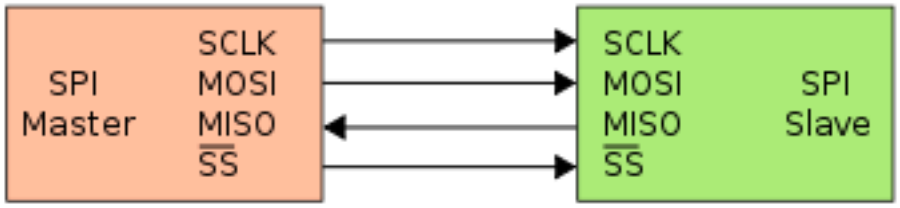
```
059   {
060     I2C_ClearITPendingBit(I2C2, I2C_IT_AF);
061   }
062 }
063
064 /**
065   * @brief  This function handles I2Cx event interrupt request.
066   * @param  None
067   * @retval None
068   */
069 volatile uint32_t Event = 0;
070 volatile uint8_t Tx_Idx;
071 volatile uint8_t Rx_Idx;
072 extern uint8_t SlaveTxBuffer[];
073 extern uint8_t SlaveRxBuffer[];
074 void I2C2_EV_IRQHandler(void)
075 {
076   /* Get Last I2C Event */
077   Event = I2C_GetLastEvent(I2C2);
078   switch (Event)
079   {
080                                                                                 /*
********************************************************************************/
081   /*                    Slave Transmitter Events                    */
082   /*                                                  */
083                                                                                 /*
********************************************************************************/
084
085   /* Check on EV1 */
086   case I2C_EVENT_SLAVE_TRANSMITTER_ADDRESS_MATCHED:
087     Tx_Idx = 0;
088     I2C_ITConfig(I2C2, I2C_IT_BUF , ENABLE);
089     break;
090   /* Check on EV3 */
```

```
case I2C_EVENT_SLAVE_BYTE_TRANSMITTING:
case I2C_EVENT_SLAVE_BYTE_TRANSMITTED:
 I2C_SendData(I2C2, SlaveTxBuffer[Tx_Idx++]);
 break;

/****************************************************************************/
/*                    Slave Receiver Events                    */
/*                                                             */
/****************************************************************************/

/* check on EV1*/
case I2C_EVENT_SLAVE_RECEIVER_ADDRESS_MATCHED:
 Rx_Idx = 0;
 break;

/* Check on EV2*/
case I2C_EVENT_SLAVE_BYTE_RECEIVED:
case (I2C_EVENT_SLAVE_BYTE_RECEIVED | I2C_SR1_BTF):
 SlaveRxBuffer[Rx_Idx++] = I2C_ReceiveData(I2C2);
 break;

/* Check on EV4 */
case I2C_EVENT_SLAVE_STOP_DETECTED:
 I2C_GetFlagStatus(I2C2, I2C_FLAG_STOPF);
 I2C_Cmd(I2C2, ENABLE);
 break;

default:
 break;
}
}
```

3.6. Lập trình SPI

3.6.1. Giới thiệu giao diện SPI

SPI (Serial Peripheral Interface, SPI bus — Giao diện Ngoại vi Nối tiếp) là một chuẩn đồng bộ nối tiếp để truyền dữ liệu ở chế độ song công toàn phần full-duplex (hai chiều, hai phía), do công ty Motorola thiết kế nhằm đảm bảo sự liên hợp giữa các vi điều khiển và thiết bị ngoại vi một cách đơn giản và giá rẻ. SPI còn được gọi là giao diện **bốn-dây** (four-wire).



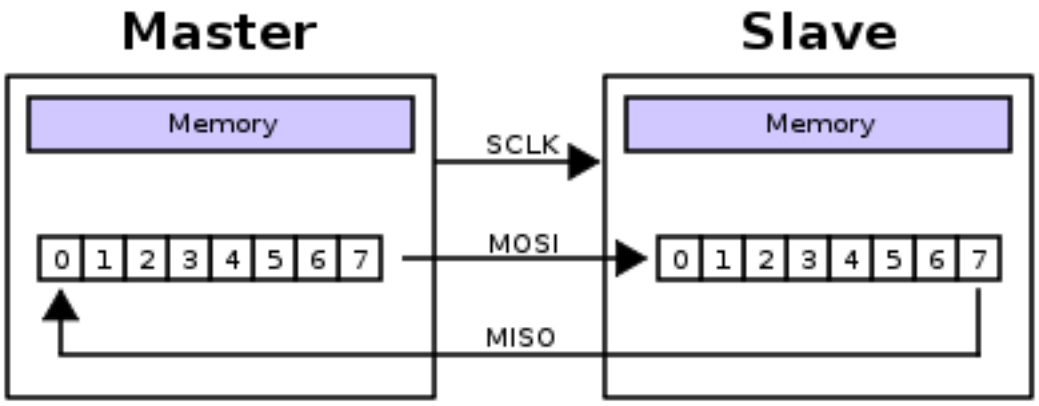
Hình 3.9. Giao diện SPI 4 dây

Trong giao diện SPI có sử dụng bốn tín hiệu số:

- MOSI hay SI – Cổng ra của bên chủ động, cổng vào của bên bị động (Master Out Slave In), dành cho việc truyền dữ liệu từ thiết bị chủ động đến thiết bị bị động.
- MISO hay SO – Cổng vào của bên chủ động, cổng ra của bên bị động (Master In Slave Out), dành cho việc truyền dữ liệu từ thiết bị bị động đến thiết bị chủ động.
- SCLK hay SCK — tín hiệu đồng nối tiếp (Serial Clock), dành cho việc truyền tín hiệu đồng cho thiết bị bị động.
- CS hay SS — chọn vi mạch, chọn bên bị động (Chip Select, Slave Select).

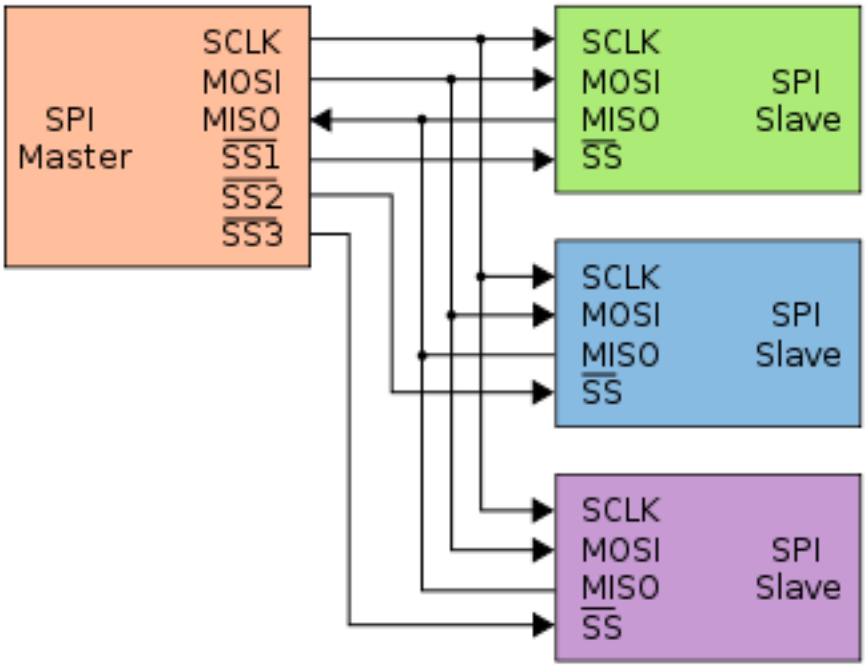
Trong đó nếu là chip master sẽ có quyền quyết định xung nhịp SCK (hay chủ động nguồn xung nhịp SCK).

Ở chế độ song công toàn phần thì trong cùng một thời điểm cả hai thiết bị đều có thể phát và nhận dữ liệu, đây là một ưu thế rất lớn của chuẩn truyền thông này.



Hình 3.10. Chế độ song công toàn phần

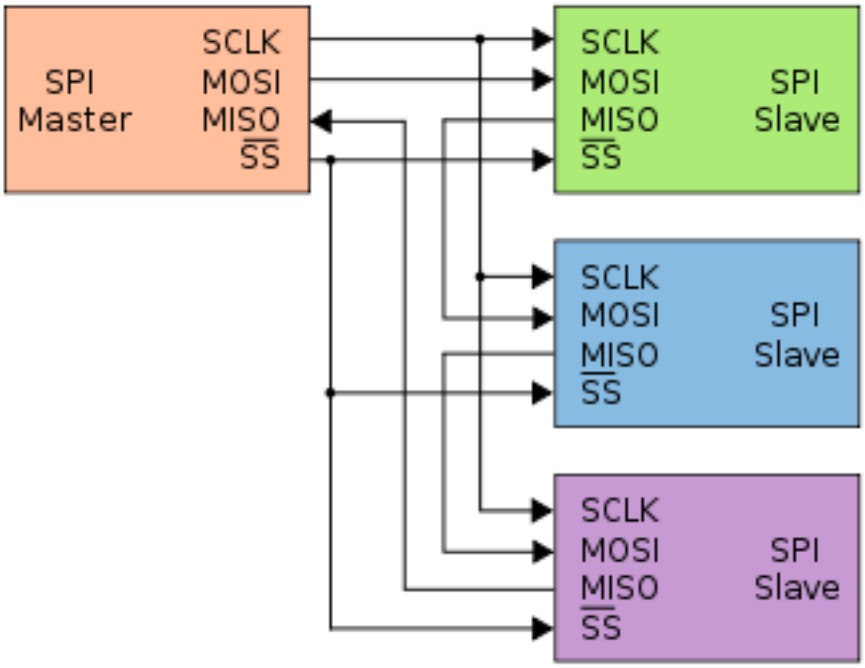
Ngoài ra chip master có thể giao tiếp với nhiều chip slave trong cùng mạng và có nhiều phương pháp khác nhau.



Hình 3.11. Một chíp master SPI có thể giao tiếp với nhiều chip slave SPI

Ở phương pháp này chip master cần nhiều đường SS, có thể thay thế bằng đường IO thông thường. Trong một thời điểm chỉ nên giao tiếp với một chip slave để tránh trường hợp các chip slave đẩy dữ liệu về cùng lúc sẽ gây lỗi dữ liệu trên đường MISO.

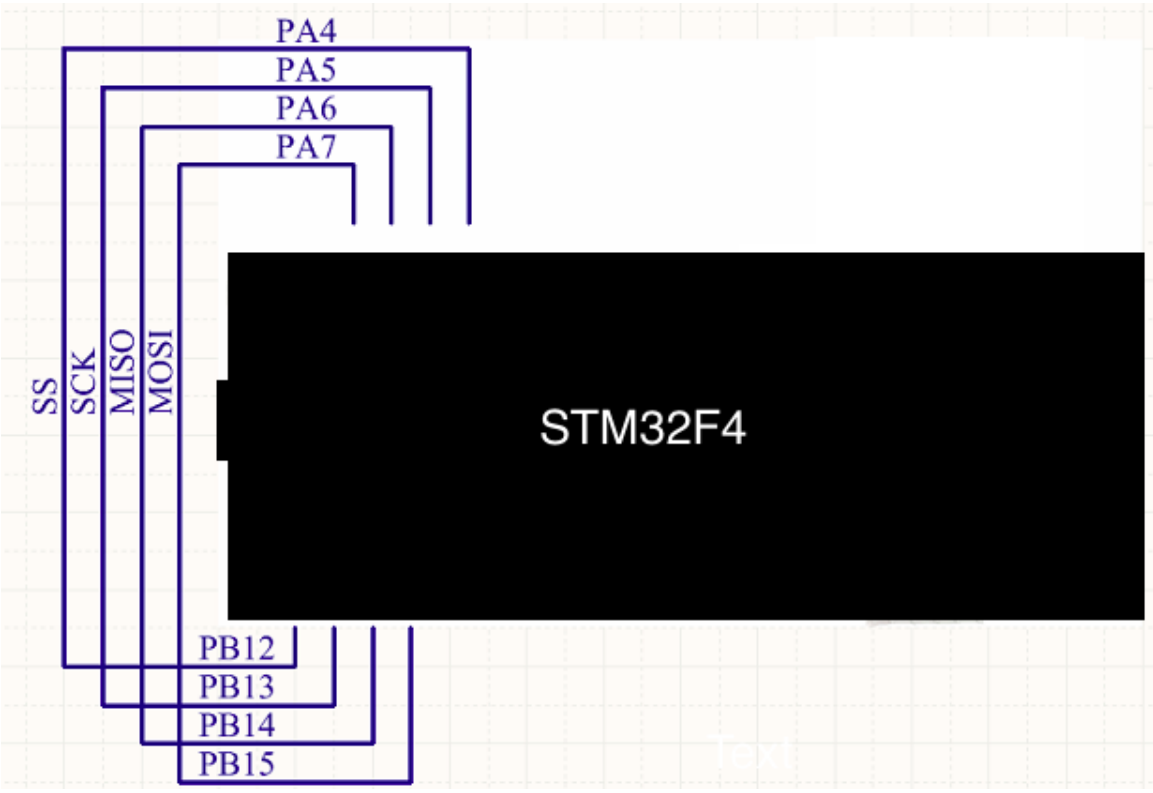
Thí dụ sử dụng phương pháp như hình bên dưới được gọi là "Daisy chain":



Hình 3.12. Phương pháp giao tiếp Daisy Chain trong chuẩn SPI

3.6.2. Kết nối phần cứng

Kết nối phần cứng như sơ đồ bên dưới:



Hình 3.13. Thí dụ kết nối phần cứng trên SPI

3.6.3. Lập trình SPI cơ bản

Thí dụ này sử dụng 2 bộ SPI, một cho master và một cho slave. Phần master thì sử dụng ở chế độ thông thường, không ngắt, không DMA. Riêng slave sử dụng ngắt nhận, do thời điểm truyền dữ liệu của master là không biết trước nên cần dùng ngắt để tránh mất dữ liệu.

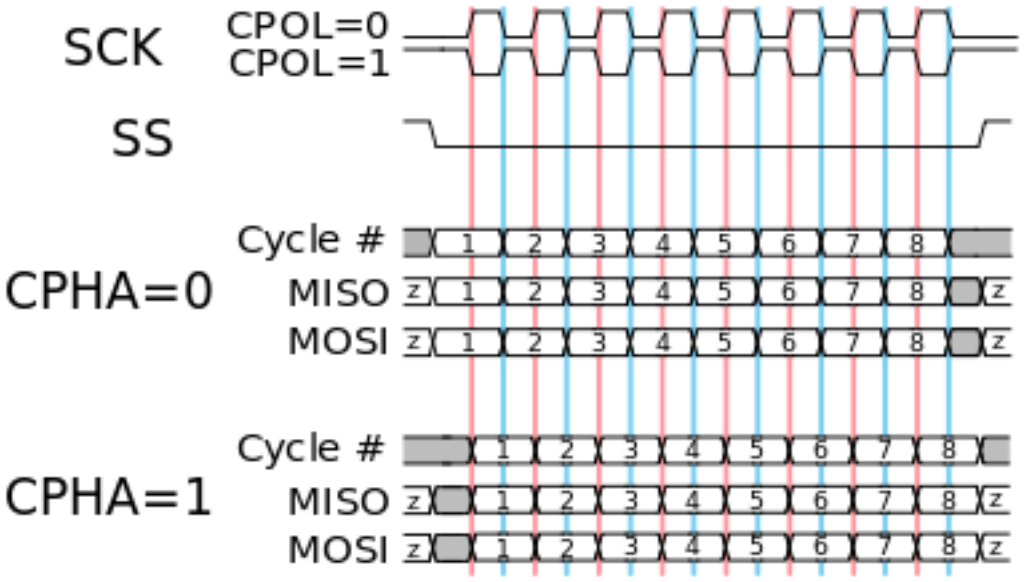
Giải thích hàm SPI_Configuration(), trong hàm này thiết lập SPI cho cả master và slave.

Dòng 54 : chọn chế độ truyền dữ liệu song công full-duplex, điều này có thể không cần thiết do trong thí dụ này chỉ dùng master để truyền mà không nhận dữ liệu, nhưng người lập trình cũng có thể để nguyên như trên và không quan tâm đến chế độ nhận.

Dòng 55: Chọn mode master cho SPI1.

Dòng 56: Chọn kích thước của khung truyền là 8bit.

Dòng 57-58: Ở hai dòng lệnh này cần tham khảo hình bên dưới để nắm rõ dữ liệu được lấy mẫu ở cạnh và pha như thế nào.



Hình 3.14. Lấy mẫu dữ liệu ở cạnh và pha

Dòng 59: Chọn sử dụng chân SS bằng phần mềm, trên thực tế có thể không

cần quan tâm do bên trên đã khai báo một chân IO (PA4) làm chức năng SS.

Dòng 60: Bộ chia tần số của SPI, sau khi qua bộ chia này ta sẽ suy ra được xung nhịp trên chân SCK. Do ta sử dụng SPI1 thuộc khối APB2 nên tốc độ tối đa mà ngoại vi trên khối này đạt được là 84MHz. Ởđây chọn bộ chia 256 vậy tần số tối đa mà chân SCK đạt được là 84MHz/256 = 328.125KHz.

Dòng 61 : bit truyền đi đầu tiên là bit MSB có nghĩa là bit có trọng số cao nhất bit thứ 8.

Ở phần khai báo cho slave cũng tương tự như khai báo cho master, nhưng mode hoạt động là mode slave và thiết lập thêm ngắt khi quá trình nhận dữ liệu hoàn tất như dòng 95.

Kết thúc hàm này bạn cho cả hai khối SPI này hoạt động.

Ở chương trình chính dòng 25 sử dụng 1 macro SS_DIS đã được tôi định nghĩa ở dòng 9 để tiện lợi nếu thay đổi các chân IO ta chỉ việc thay đổi bên trên mà không cần thay đổi toàn bộ chương trình của bạn đang viết.

Dòng 26: Sử dụng hàm SPI_I2S_SendData() để gửi dữ liệu từ master lên đường truyền SPI, đối số SPI_data_send chính là biến dữ liệu cần truyền đi.
Dòng 27: Vòng lặp này để bẫy chương trình, khi kết thúc quá trình truyền thì sẽ thoát khỏi vòng lặp. Việc này để tránh trường hợp tốc độ vi điều khiển quá nhanh, dữ liệu cũ còn chưa truyền đi hết thì dữ liệu mới đã được gửi tiếp.

Mục đích sử dụng delay ở dùng 29 là để trì hoãn khoản mức cao mà master tạo ra trên SS, nếu không trì hoãn thì chương trình quay lại dòng 25 làm slave chưa kịp phát hiện ra tín hiệu mức cao.

Khi có sự kiện nhận dữ liệu trên slave lặp tức chương trình sẽ nhảy đến hàm ngắt SPI2_IRQHandler() ở dòng 59 trong file stm32f4xx_it.c.
Giải thích trên file stm32f4xx_it.c:

Ở dòng 4 khai báo thêm từ khóa extern để main.c và stm32f4xx_it.c sử dụng chung biến SPI_data_get.

Dòng 63: Khi có một tín hiệu ngắt do quá trình nhận đã hoàn tất tôi sẽ dùng hàm SPI_I2S_ReceiveData() lấy dữ liệu trong thanh ghi ra và gán vào biến

SPI_data_get.

File main.c

```
001  #include "stm32f4xx.h"
002
003  /*      MASTER              SLAVE
004    PA4 --- CONTROL_SS    PB12 --- SPI2_SS
005    PA5 --- SPI1_SCK      PB13 --- SPI2_SCK
006    PA6 --- SPI1_MISO     PB14 --- SPI2_MISO
007    PA7 --- SPI1_MOSI     PB15 --- SPI2_MOSI
008  */
009  #define SS_DIS GPIO_ResetBits(GPIOA, GPIO_Pin_4)
010  #define SS_EN  GPIO_SetBits(GPIOA, GPIO_Pin_4)
011
012  volatile uint8_t SPI_data_send,SPI_data_get;
013  GPIO_InitTypeDef  GPIO_InitStructure;
014  SPI_InitTypeDef   SPI_InitStructure;
015  NVIC_InitTypeDef  NVIC_InitStructure;
016
017  void SPI_Configuration(void);
018  void Delay(__IO uint32_t nCount);
019
020  int main(void)
021  {
022    SPI_Configuration();
023    while (1)
024    {
025      SS_DIS;
026      SPI_I2S_SendData(SPI1, SPI_data_send);
027      while(SPI_I2S_GetFlagStatus(SPI1,SPI_I2S_FLAG_TXE)==RESET);
028      SS_EN;
029      Delay(1000);
030    }
031  }
032
033  void SPI_Configuration(void)
034  {
035    /* SPI_MASTER configuration ------------------------------------------------*/
036    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
037    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
038
039    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
040    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
041    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
042    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
043    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
044    GPIO_Init(GPIOA, &GPIO_InitStructure);
045    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
046    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
047    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
```

```
048   GPIO_Init(GPIOA, &GPIO_InitStructure);
049
050   GPIO_PinAFConfig(GPIOA,GPIO_PinSource5,GPIO_AF_SPI1);
051   GPIO_PinAFConfig(GPIOA,GPIO_PinSource6,GPIO_AF_SPI1);
052   GPIO_PinAFConfig(GPIOA,GPIO_PinSource7,GPIO_AF_SPI1);
053
054   SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
055   SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
056   SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
057   SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
058   SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
059   SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
060   SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_256;
061   SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
062   SPI_InitStructure.SPI_CRCPolynomial = 7;
063   SPI_Init(SPI1, &SPI_InitStructure);
064
065   /* SPI_SLAVE configuration ------------------------------------------------*/
066   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
067   RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);
068       GPIO_InitStructure.GPIO_Pin  =  GPIO_Pin_12  |  GPIO_Pin_13  |  GPIO_Pin_14  |
GPIO_Pin_15;
069   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
070   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
071   GPIO_Init(GPIOB, &GPIO_InitStructure);
072
073   GPIO_PinAFConfig(GPIOB,GPIO_PinSource12,GPIO_AF_SPI2);
074   GPIO_PinAFConfig(GPIOB,GPIO_PinSource13,GPIO_AF_SPI2);
075   GPIO_PinAFConfig(GPIOB,GPIO_PinSource14,GPIO_AF_SPI2);
076   GPIO_PinAFConfig(GPIOB,GPIO_PinSource15,GPIO_AF_SPI2);
077
078   SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
079   SPI_InitStructure.SPI_Mode = SPI_Mode_Slave;
080   SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
081   SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
082   SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
083   SPI_InitStructure.SPI_NSS = SPI_NSS_Hard;
084   SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;
085   SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
086   SPI_InitStructure.SPI_CRCPolynomial = 7;
087   SPI_Init(SPI2, &SPI_InitStructure);
088
089   NVIC_InitStructure.NVIC_IRQChannel = SPI2_IRQn;
090   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
091   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
092   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
093   NVIC_Init(&NVIC_InitStructure);
094
095   SPI_I2S_ITConfig(SPI2, SPI_I2S_IT_RXNE, ENABLE);
096
097   /* Enable SPI_SLAVE */
```

```
098   SPI_Cmd(SPI2, ENABLE);
099   /* Enable SPI_MASTER */
100   SPI_Cmd(SPI1, ENABLE);
101 }
102
103 void Delay(__IO uint32_t nCount)
104 {
105   while(nCount--)
106   {
107   }
108 }
109
110 #ifdef  USE_FULL_ASSERT
111 void assert_failed(uint8_t* file, uint32_t line)
112 {
113   /* User can add his own implementation to report the file name and line number,
114      ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
115
116   /* Infinite loop */
117   while (1)
118   {
119   }
120 }
121 #endif
```

File stm32f4xx_it.c

```
01 #include "stm32f4xx_it.h"
02 #include "stm32f4xx.h"
03
04 extern uint8_t SPI_data_get;
05
06 void NMI_Handler(void)
07 {
08 }
09
10 void HardFault_Handler(void)
11 {
12   /* Go to infinite loop when Hard Fault exception occurs */
13   while (1)
14   {
15   }
16 }
17
18 void MemManage_Handler(void)
19 {
20   /* Go to infinite loop when Memory Manage exception occurs */
21   while (1)
22   {
23   }
24 }
```

```
25
26  void BusFault_Handler(void)
27  {
28    /* Go to infinite loop when Bus Fault exception occurs */
29    while (1)
30    {
31    }
32  }
33
34  void UsageFault_Handler(void)
35  {
36    /* Go to infinite loop when Usage Fault exception occurs */
37    while (1)
38    {
39    }
40  }
41
42  void SVC_Handler(void)
43  {
44  }
45
46  void DebugMon_Handler(void)
47  {
48  }
49
50  void PendSV_Handler(void)
51  {
52  }
53
54  void SysTick_Handler(void)
55  {
56
57  }
58
59  void SPI2_IRQHandler(void)
60  {
61    if (SPI_I2S_GetITStatus(SPI2, SPI_I2S_IT_RXNE) != RESET)
62    {
63      SPI_data_get = SPI_I2S_ReceiveData(SPI2);
64      //SPI_I2S_ClearFlag(SPI2, SPI_I2S_IT_RXNE);
65    }
66  }
```

TÀI LIỆU THAM KHẢO

[1] Ngô Thị Vinh, *Tập bài giảng Lập trình nhúng nâng cao*, Bộ môn Công nghệ Kỹ thuật máy tính (2014)

[2]. Hitex.com, The insider's guide to STM32, ARM7 based microcontroller.

[3]. Geoffrey Brown (2014), Discovering the STM32 Microcontroller

[4]. www.st.com, Cortex-M3 programming manual.

[5]. www.st.com (2013), *Fun, easy introduction kit for STM32 microcontrollers*