

ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA ĐIỆN – ĐIỆN TỬ

-----o0o-----



BÀI TẬP LỚN

ĐỀ TÀI:
TÌM HIỂU VỀ CHẾ ĐỘ ADC VÀ USART TRÊN
KIT RENESAS RA6M5

Giảng viên hướng dẫn: Nguyễn Phan Hải Phú

Họ và tên	MSSV
Lý Thị Huỳnh Như	2114336
Nguyễn Tấn Tài	2212991
Văn Mỹ Trân	2012261

TP. HỒ CHÍ MINH, THÁNG 12 NĂM 2025

1.1 Các chế độ của giao tiếp SPI.

Giao tiếp SPI của vi điều khiển RA6M5 được chia thành 2 chế độ chính gồm chế độ hoạt động SPI (SPI Operation) và chế độ đồng bộ xung nhịp (Clock Synchronous Operation).

1.1.1. Chế độ hoạt động SPI (SPI Operation).

Ở chế độ này, RA6M5 sử dụng đầy đủ 4 dây tín hiệu gồm MOSI, MISO, SSL và RSPCK, để giao tiếp với các thiết bị khác.

a. Chế độ đơn master (Single master mode) và đa master (Multi-master mode).

Hai chế độ này chỉ có một điểm khác nhau duy nhất ở việc sử dụng chế độ phát hiện lỗi chế độ (mode fault error), chế độ này chỉ được sử dụng ở đa master. Để thiết lập chế độ đơn hoặc đa master cho vi điều khiển RA6M5, ta cần truy cập vào các bit MSTR, MODFEN, SPMS trong thanh ghi SPCR như sau:

- MSTR = 1, SPI hoạt động với vai trò master.
- MODFEN = 0, đối với đơn master hoặc MODFEN = 1, đối với đa master.
- SPMS = 0, chọn chế độ hoạt động SPI (giao tiếp 4 dây).

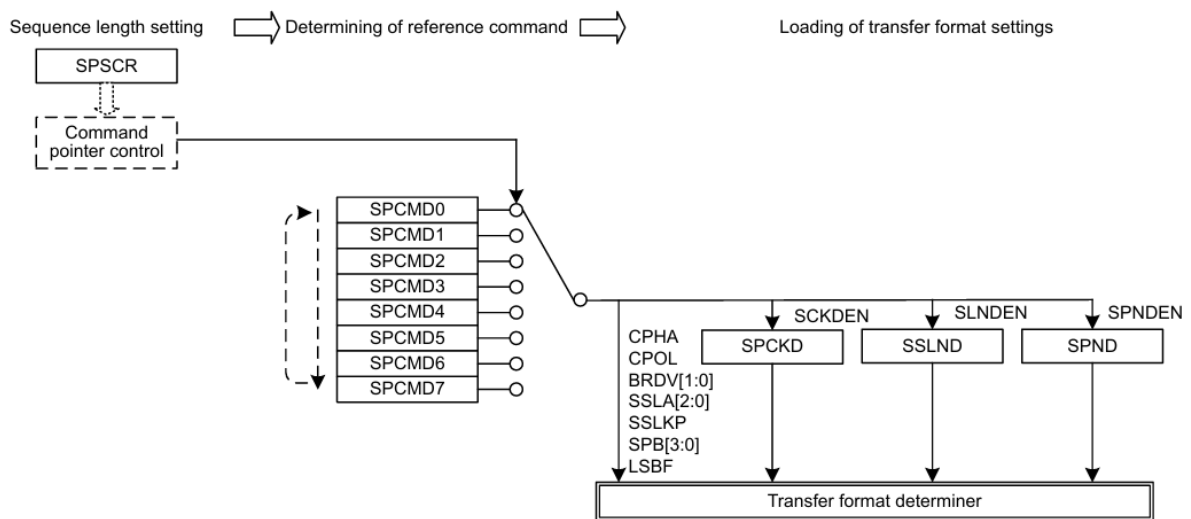
Trong chế độ này, MCU hoàn toàn kiểm soát giao tiếp, nó tạo ra xung nhịp từ chân RSPCK và chủ động điều khiển tín hiệu SSLn (mức dữ liệu đầu ra của các chân này được thiết lập bằng các thanh ghi SSLP) để chọn một thiết bị slave. Chuẩn bị truyền dữ liệu, phần mềm sẽ ghi dữ liệu vào thanh ghi SPDR/SPDR_HA/SPDR_BY, dữ liệu này sẽ được cập nhật vào bộ đệm truyền SPTX với điều kiện bộ đệm truyền trống (cờ SPSR.STEF = 0), dữ liệu cho lần truyền tiếp theo chưa được đặt. Tiếp theo, nếu thanh ghi dịch trống và số lượng khung truyền được đặt trong bit SPCR.SPFC[1:0] đã được hoàn thành, dữ liệu từ bộ đệm SPTX được chuyển vào thanh ghi dịch để chuẩn bị truyền. Khi đó, trạng thái của thanh ghi dịch được đặt thành trạng thái đầy và bắt đầu truyền dữ liệu ra ngoài, khi truyền xong, trạng thái sẽ được đặt lại về trạng thái trống.

Quá trình truyền dữ liệu sẽ kết thúc khi một cạnh xung cuối cùng của thời gian lấy mẫu được gửi đi từ RSPCK. Thời gian lấy mẫu cuối cùng phụ thuộc vào độ dài dữ liệu truyền được quyết định bởi bit SPCMDm.SPB[3:0] ở phía master. Còn ở phía slave, dữ liệu nhận được sẽ nằm trong thanh ghi dịch, nếu bộ đệm nhận SPRX trống (cờ

SPSR.SPRF = 0), dữ liệu sẽ được sao chép vào bộ đệm và sẽ được phần mềm đọc thông qua thanh ghi SPDR/SPDR_HA/SPDR_BY.

RA6M5 còn hỗ trợ cơ chế điều khiển chuỗi (Sequence control), một chức năng giúp master có thể truyền tải với nhiều định dạng khác nhau một cách tuần tự và tự động. Với các thanh ghi xác định định dạng truyền tải như SPSCR, SPBR, SPCKD, SSLND, SPND và SPCMDm, mỗi thanh ghi đảm nhận một vai trò cụ thể. Thanh ghi SPSCR xác định cấu hình chung của chuỗi. Thanh ghi SPBR quyết định một vài cài đặt về tốc độ bit, bao gồm SPCKD (độ trễ xung clock), SSLND (độ trễ phủ định các chân SSL) và SPND (độ trễ lần truy cập tiếp theo). Còn thanh ghi SPCMDm sẽ quyết định các mục sau:

- Giá trị tín hiệu đầu ra của các chân SSLni.
- MSB-first hay LSB-first.
- Độ dài dữ liệu.
- Một vài cài đặt về tốc độ bit.
- Cực tính và pha của RSPCK.
- Quyết định liệu có cho phép SPCKD, SSLND hoặc SPND được tham chiếu hay không.



Hình trên mô tả cơ chế tự động thực hiện một chuỗi các lệnh được theo các bước như sau:

- Độ dài các chuỗi lệnh được thiết lập trong thanh ghi SPSCR và các lệnh này được lưu vào một phần hoặc toàn bộ các thanh ghi SPCMDm (SPCMD0, SPCMD1,...).

- SPI sử dụng một con trỏ là các bit SPSSR.SPCP[2:0] để trỏ đến thanh ghi lưu các lệnh SPCMDm.
- Khi kích hoạt chức năng SPI, con trỏ được đặt về lệnh đầu tiên (SPCMD0) và các lệnh về định dạng truyền được áp dụng cho lần truyền đầu tiên.
- Sau mỗi lần truyền hoặc sau mỗi lần kết thúc chu kỳ làm trễ cho lần truy cập tiếp theo (SPND), SPI tăng con trỏ đến lệnh kế tiếp.
- Lặp lại cho đến khi hoàn thành lệnh cuối cùng (SPCMDm), con trỏ sẽ được đặt lại về vị trí ban đầu và tiếp tục truyền theo các chuỗi lệnh.

b. Chế độ slave (Slave mode).

Khác với hai chế độ trên, để cấu hình chế độ slave cho RA6M5, ta ghi vào các bit MSTR, MODFEN và SPMS như sau:

- MSTR = 0, MCU hoạt động với vai trò slave.
- MODFEN tùy vào ứng dụng, người có thể lựa chọn bật hoặc không bật chế độ phát hiện lỗi chế độ.
- SPMS = 0, chọn chế độ hoạt động SPI (giao tiếp 4 dây).

MCU sẽ hoạt động một cách thụ động, nó không tạo ra xung nhịp mà nhận xung nhịp từ master, đôi khi master kích hoạt chân SSLn0 của slave thì quá trình truyền nhận mới bắt đầu. Nếu bit SPCMD0.CPHA = 0, nếu SPI phát hiện chân SSLn0 được kích hoạt, slave phải ngay lập tức truyền dữ liệu hợp lệ ra chân MISO. Do đó, khi CPHA = 0, việc SSLn0 được kích hoạt sẽ khởi động quá trình truyền tải dữ liệu. Nếu CPHA = 1, nếu SPI phát hiện cạnh xung RSPCK đầu tiên khi SSLn0 đã được kích hoạt, quá trình truyền nhận sẽ bắt đầu.

Quá trình truyền nhận sẽ kết thúc khi SPI phát hiện cạnh xung RSPCK tương ứng với thời gian lấy mẫu cuối cùng. Khi đó, nếu bộ đệm nhận SPRX trống (cờ SPSR.SPRF = 0) SPI sao chép dữ liệu nhận được từ thanh ghi dịch vào bộ đệm nhận. Trong quá trình truyền nhận, nếu slave phát hiện chân SSLn0 ở trạng thái không kích hoạt, lỗi chế độ (mode fault) sẽ xảy ra. Cơ chế phát hiện lỗi này chỉ được sử dụng nếu bit MODFEN = 1. Lưu ý khi MCU hoạt động ở chế độ đơn slave, thông thường ở chế độ này, chân SSLn0 luôn được đặt ở trạng thái kích hoạt. Nếu ở slave, bit CPHA = 0, quá trình truyền nhận chỉ bắt đầu khi cạnh xung kích hoạt SSLn0. Từ đó dẫn đến việc slave không thể phát hiện

cạnh xung kích hoạt SSLn0 và không thể bắt đầu truyền nhận. Vì vậy, không nên cố định mức tín hiệu của SSLn0 nếu bit CPHA = 0 ở slave hoặc cần phải đặt CPHA = 1 nếu muốn cố định SSLn0.

1.2.1 Chế độ đồng bộ xung nhịp (Clock Synchronous Operation).

Ở chế độ hoạt động xung nhịp chỉ sử dụng 3 dây tín hiệu gồm RSPCK, MOSI và MISO, các chân SSL0 đến SSL3 không được sử dụng mà được giải phóng thành các chân IO thông thường. Lưu ý ở chế độ này, không thể phát hiện được lỗi chế độ do không sử dụng các chân SSLn.

a. Chế độ master (Master mode).

Để thiết lập chế độ master cho vi điều khiển RA6M5, ta cần truy cập vào các bit MSTR, MODFEN, SPMS trong thanh ghi SPCR như sau:

- MSTR = 1, SPI hoạt động với vai trò master.
- MODFEN = 0, vì chế độ phát hiện lỗi bị vô hiệu hóa.
- SPMS = 1, chọn chế độ đồng bộ xung nhịp (giao tiếp 3 dây).

Quá trình chuẩn bị truyền dữ liệu ở chế độ master tương tự như ở giao tiếp 4 dây được mô tả ở trên. Cụ thể, SPI sẽ cập nhật dữ liệu được ghi vào thanh ghi SPDR/SPDR_HA/SPDR_BY vào bộ đệm truyền SPTX với điều kiện bộ đệm SPTX trống (cờ SPSR.SPTEF = 0) và dữ liệu cho lần truyền tiếp theo chưa được đặt. Khi thanh ghi dịch trống và số khung truyền đã được đặt bằng bit SPCR.SPFC[1:0], dữ liệu này sẽ được nạp vào thanh ghi dịch để chuẩn bị truyền đi. Sau khi truyền xong, thanh ghi dịch sẽ chuyển về trạng thái trống. Điểm khác biệt so với chế độ master ở trên chính là quá trình truyền nhận được tiến hành mà không cần sử dụng các chân SSLn0.

Quá trình truyền nhận sẽ kết thúc khi cạnh xung RSPCK tương ứng với thời điểm lấy mẫu cuối cùng được phát hiện. Sau khi kết thúc, nếu bộ đệm nhận SPRX của slave trống, dữ liệu từ thanh ghi dịch sẽ được sao chép vào bộ đệm và sẽ được đọc thông qua thanh ghi SPDR/SPDR_HA/SPDR_BY. Độ dài dữ liệu tùy thuộc vào độ dài dữ liệu truyền tải được quyết định bởi master bởi các bit SPCMDm.SPB[3:0].

Tương tự như master ở chế độ hoạt động SPI (SPI operation), ở chế độ đồng bộ xung nhịp cũng có cơ chế điều khiển chuỗi mà không cần đến các chân SSLni. Định dạng truyền tải sẽ được xác định bởi các thanh ghi SPSCR, SPCMDm, SPBR, SPCKD,

SSLND và SPND với các vai trò tương tự như ở SPI operation. Mặc dù các chân SSLn_i không được sử dụng về mặt vật lý nhưng các cài đặt liên quan đến chúng vẫn hợp lệ. Cơ chế tự động thực hiện một chuỗi các lệnh được theo các bước như sau:

- Độ dài các chuỗi lệnh được thiết lập trong thanh ghi SPSCR và các lệnh này được lưu vào một phần hoặc toàn bộ các thanh ghi SPCMD_m (SPCMD0, SPCMD1,...).
- SPI sử dụng một con trỏ là các bit SPSSR.SPCP[2:0] để trỏ đến thanh ghi lưu các lệnh SPCMD_m.
- Khi kích hoạt chức năng SPI, con trỏ được đặt về lệnh đầu tiên (SPCMD0) và các lệnh về định dạng truyền được áp dụng cho lần truyền đầu tiên.
- Sau mỗi lần truyền hoặc sau mỗi lần kết thúc chu kỳ làm trễ cho lần truy cập tiếp theo (SPND), SPI tăng con trỏ đến lệnh kế tiếp.
- Lặp lại cho đến khi hoàn thành lệnh cuối cùng (SPCMD_m), con trỏ sẽ được đặt lại về vị trí ban đầu và tiếp tục truyền theo các chuỗi lệnh.

b. Chế độ slave (Slave mode).

Để thiết lập chế master cho vi điều khiển RA6M5, ta cần truy cập vào các bit MSTR, MODFEN, SPMS trong thanh ghi SPCR như sau:

- MSTR = 0, SPI hoạt động với vai trò slave.
- MODFEN = 0, vì chế độ phát hiện lỗi bị vô hiệu hóa.
- SPMS = 1, chọn chế độ đồng bộ xung nhịp (giao tiếp 3 dây).

Khi cạnh xung RSPCK_n đầu tiên sẽ kích hoạt quá trình truyền nhận nối tiếp, dữ liệu ngay lập tức được chuyển ra chân MISO. Khi phát hiện cạnh xung RSPCK_n tương ứng với thời điểm lấy mẫu cuối cùng, quá trình truyền nhận sẽ kết thúc. Dữ liệu nhận được trong thanh ghi dịch sẽ được cập nhật vào bộ đệm nhận SPRX nếu bộ đệm trống (cờ SPSR.SPRF = 0) và sẽ được đọc thông qua thanh ghi SPDR/SPDR_HA/SPDR_BY. Độ dài dữ liệu được quyết định bởi master thông qua các bit SPCMD0.SPB[3:0]. Sau khi kết thúc truyền nhận, trạng thái thanh ghi dịch được đặt về trạng thái trống bất kể trạng thái hiện tại của bộ đệm.

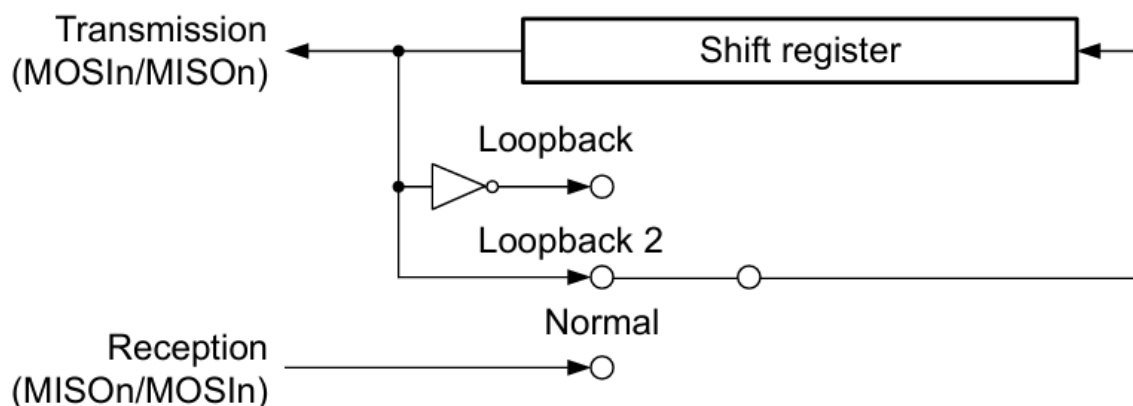
1.3.1. Chế độ Loopback.

Ở chế độ loopback, SPI sẽ ngắt kết nối giữa thanh ghi dịch và chân MISO_n nếu bit SPCR.MSTR = 1 hoặc ngắt kết nối giữa MOSI_n với thanh ghi dịch nếu bit SPCR.MSTR

= 0. Thay vào đó, kết nối đầu ra của thanh ghi dịch với đầu vào của nó, hình thành một vòng lặp kín, khi đó, dữ liệu truyền sẽ trở thành dữ liệu nhận. Chế độ này được sử dụng để kiểm tra chức năng của SPI mà không cần các thiết bị khác. Để kích hoạt chế độ này ta cần đặt bit SPPCR.SPLP2 hoặc bit SPPCR.SPLP thành 1:

SPPCR.SPLP2	SPPCR.SPLP	Mô tả
0	0	Chế độ SPI tiêu chuẩn.
0	1	Chế độ Loopback: dữ liệu nhận là dữ liệu truyền đi nhưng bị đảo bit.
1	0	Chế độ Loopback 2: dữ liệu nhận là dữ liệu truyền đi nguyên bản.
1	1	Chế độ Loopback 2: dữ liệu nhận là dữ liệu truyền đi nguyên bản.

Hình bên dưới mô tả cơ chế ngắt kết nối và chọn đầu vào cho thanh ghi dịch:



Mục đích chính của chế độ Loopback là để kiểm tra chức năng truyền nhận của SPI mà không kết nối với cần thiết bị khác. Nếu dữ liệu truyền không khớp với dữ liệu nhận trong chế độ Loopback 2, từ đó xác định lỗi xảy ra ở trong module SPI. Ngược lại nếu Loopback 2 hoạt động đúng, xác định được lỗi nằm ở kết nối bên ngoài.

1.3. Các chương trình mẫu.

1.4.1. Chế độ loopback:

Lập trình thanh ghi:

```
#include "hal_data.h"
#include "stdio.h"

#define BUFFER_SIZE 8

uint8_t tx_buffer[BUFFER_SIZE] = {0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88};
uint8_t rx_buffer[BUFFER_SIZE] = {0};

void SPI1_init(void) {
    R_MSTP->MSTPCRB &= ~(1U << 29); // MSTPB29 = 0

    R_SPI1->SPCR = 0x00; //Clear SPCR register

    R_SPI1->SPCR_b.MSTR = 1; //Master mode

    R_SPI1->SPPCR = 0x02; //Loopback 2 non_invert

    R_SPI1->SPBR = 3; //Bit rate = f(PCLKA)/(2x(n+1)x2^N), N=0 as default
    R_SPI1->SPCMD[0] = 0x0700; //CPOL=0, CPHA=0, MSB first, 8-bit

    R_SPI1->SPCR_b.SPE = 1; // SPI enable
}

void spi_loopback_test(void) {
    for (int i = 0; i < BUFFER_SIZE; i++) {
        /* Wait TX buffer empty */
        while (R_SPI1->SPSR_b.SPTEF == 0);
        /* Write data */
        R_SPI1->SPDR = tx_buffer[i];
        /* Wait RX completed */
        while (R_SPI1->SPSR_b.SPRF == 0); // SPRF = 1
        /* Read data */
        rx_buffer[i] = (uint8_t)R_SPI1->SPDR;
    }

    for (int i = 0; i < BUFFER_SIZE; i++) {
        if (tx_buffer[i] != rx_buffer[i]) {
            return;
        }
    }
}

void hal_entry(void) {
    /* TODO: add your own code here */
    SPI1_init();
    spi_loopback_test();

    while (1);
}

#if BSP_TZ_SECURE_BUILD
/* Enter non-secure code */
R_BSP_NonSecureEnter();
#endif
```

```

#endif
}

#if BSP_TZ_SECURE_BUILD

FSP_CPP_HEADER
BSP_CMSE_NONSECURE_ENTRY void template_nonsecure_callable ();

/* Trustzone Secure Projects require at least one nonsecure callable function in
order to build (Remove this if it is not required to build). */
BSP_CMSE_NONSECURE_ENTRY void template_nonsecure_callable ()
{
}
FSP_CPP_FOOTER

#endif

```

1.4.2. Chế độ hoạt động SPI (giao tiếp 4 dây):

Sử dụng FSP:

Cấu hình SPI1 trong FSP Configuration:

Vào stack → New Stack → Connectivity → SPI(r_spi)

Thêm Transfer Driver cho cả Transmission và Reception, enable Transfer End Interrupt.

Vào Properties của g_spi1 và thực hiện cấu hình:

- Name: g_spi1.
- Channel: 1.
- Operating Mode: Master
- Clock Phase: Data sampling on odd edge, data variation on even edge.
- Clock Polarity: Low when idle.
- Bit Order: MSB first.
- SPI Mode: SPI operation.
- Full or Transmit only mode: Full Duplex

Chọn các chân giao tiếp ở thẻ Pins → Connectivity:SPI → SPI1.

Chương trình chính:

```

#include "hal_data.h"

uint8_t tx_buf[4] = {0xAA, 0xBB, 0xCC, 0xDD};
uint8_t rx_buf[4] = {0};
volatile bool spi_done = false;

void spi_callback(spi_callback_args_t * p_args)
{

```

```

    if (p_args->event == SPI_EVENT_TRANSFER_COMPLETE)
    {
        spi_done = true;
    }
}

void hal_entry(void)
{
    /* TODO: add your own code here */
    fsp_err_t err;
    err = R_SPI_Open(&g_spi1_ctrl, &g_spi1_cfg);
    assert(FSP_SUCCESS == err);

    err = R_SPI_WriteRead(&g_spi1_ctrl, tx_buf, rx_buf, sizeof(tx_buf),
SPI_BIT_WIDTH_8_BITS);
    assert(FSP_SUCCESS == err);

    R_BSP_SoftwareDelay(100, BSP_DELAY_UNITS_MILLISECONDS);

    err = R_SPI_Close(&g_spi1_ctrl);
    assert(FSP_SUCCESS == err);

    while (1)
    {
        R_BSP_SoftwareDelay(1000, BSP_DELAY_UNITS_MILLISECONDS);
    }

#if BSP_TZ_SECURE_BUILD
    /* Enter non-secure code */
    R_BSP_NonSecureEnter();
#endif

#if BSP_TZ_SECURE_BUILD

FSP_CPP_HEADER
BSP_CMSE_NONSECURE_ENTRY void template_nonsecure_callable ();

/* Trustzone Secure Projects require at least one nonsecure callable function in
order to build (Remove this if it is not required to build). */
BSP_CMSE_NONSECURE_ENTRY void template_nonsecure_callable ()
{

}

FSP_CPP_FOOTER

#endif

```

Lập trình thành ghi:

```

#include "hal_data.h"
#include "bsp_api.h"

#define BUFFER_SIZE 4
uint8_t tx_buffer[BUFFER_SIZE] = {0xAA, 0xBB, 0xCC, 0xDD};
uint8_t rx_buffer[BUFFER_SIZE] = {0};

void spi_pins_init(void) {

```

```

R_PMISC->PWPR_b.B0WI = 0;
R_PMISC->PWPR_b.PFSWE = 1;

R_PFS->PORT[4].PIN[10].PmnPFS_b.PSEL = 0b00110; // SPI1_MISOB
R_PFS->PORT[4].PIN[11].PmnPFS_b.PSEL = 0b00110; // SPI1_MOSIB
R_PFS->PORT[4].PIN[12].PmnPFS_b.PSEL = 0b00110; // SPI1_RSCKB
R_PFS->PORT[4].PIN[13].PmnPFS_b.PSEL = 0b00110; // SPI1_SSIB0

R_PFS->PORT[4].PIN[10].PmnPFS_b.PDR = 0;
R_PFS->PORT[4].PIN[11].PmnPFS_b.PDR = 0;
R_PFS->PORT[4].PIN[12].PmnPFS_b.PDR = 0;
R_PFS->PORT[4].PIN[13].PmnPFS_b.PDR = 0;

R_PFS->PORT[4].PIN[10].PmnPFS_b.PCR = 0;
R_PFS->PORT[4].PIN[11].PmnPFS_b.PCR = 0;
R_PFS->PORT[4].PIN[12].PmnPFS_b.PCR = 0;
R_PFS->PORT[4].PIN[13].PmnPFS_b.PCR = 0;

R_PFS->PORT[4].PIN[10].PmnPFS_b.PODR = 0;
R_PFS->PORT[4].PIN[11].PmnPFS_b.PODR = 0;
R_PFS->PORT[4].PIN[12].PmnPFS_b.PODR = 0;
R_PFS->PORT[4].PIN[13].PmnPFS_b.PODR = 0;

R_PMISC->PWPR_b.PFSWE = 0;
R_PMISC->PWPR_b.B0WI = 1;
}

void spi_init(void) {
    R_MSTP->MSTPCRB &= ~(1U << 29); // MSTPB29 = 0

    R_SPI1->SPCR = 0x00; //Clear SPCR register

    R_SPI1->SPCR_b.SPMS = 0; //SPI operation
    R_SPI1->SPCR_b.TXMD = 0; //Full-duplex
    R_SPI1->SPCR_b.MODFEN = 0; //No mode fault detect
    R_SPI1->SPCR_b.MSTR = 1; //Master mode

    R_SPI1->SPCMD[0] = 0x0700; //CPOL=0, CPHA=0, MSB first, 8-bit
    R_SPI1->SPBR = 4; //Bit rate = f(PCLKB)/(2x(n+1)x2^N) = 5Mbps (N
depend on BRDV bit)

    R_SPI1->SPCR_b.SPE = 1; // SPI enable
}

void spi_operation(void) {
    for (int i = 0; i < BUFFER_SIZE; i++) {
        /* Wait TX buffer empty */
        while (R_SPI1->SPSR_b.SPTEF == 0);

        /* Write data */
        R_SPI1->SPDR = tx_buffer[i];

        /* Wait RX completed */
        while (R_SPI1->SPSR_b.SPRF == 0); // SPRF = 1

        /* Read data */
        rx_buffer[i] = (uint8_t)R_SPI1->SPDR;
    }
}

```

```

    }
}
void hal_entry(void)
{
    /* TODO: add your own code here */
    spi_pins_init();
    spi_init();
    spi_operation();

    while (1);

#ifdef BSP_TZ_SECURE_BUILD
    /* Enter non-secure code */
    R_BSP_NonSecureEnter();
#endif
}

#ifdef BSP_TZ_SECURE_BUILD

FSP_CPP_HEADER
BSP_CMSE_NONSECURE_ENTRY void template_nonsecure_callable ();

/* Trustzone Secure Projects require at least one nonsecure callable function in
order to build (Remove this if it is not required to build). */
BSP_CMSE_NONSECURE_ENTRY void template_nonsecure_callable ()
{

}

FSP_CPP_FOOTER

#endif

```

1.4.3. Chế độ đồng bộ xung nhịp (giao tiếp 3 dây):

Sử dụng FSP:

Cấu hình SPI1 trong FSP Configuration:

Vào stack → New Stack → Connectivity → SPI(r_spi)

Thêm Transfer Driver cho cả Transmission và Reception, enable Transfer End Interrupt.

Vào Properties của g_spi1 và thực hiện cấu hình:

- Name: g_spi1.
- Channel: 1.
- Operating Mode: Master
- Clock Phase: Data sampling on odd edge, data variation on even edge.
- Clock Polarity: Low when idle.
- Bit Order: MSB first.

- SPI Mode: Clock Synchronous Operation.
- Full or Transmit only mode: Full Duplex

Chọn các chân giao tiếp ở thẻ Pins → Connectivity:SPI → SPI1.

Chương trình chính:

```
#include "hal_data.h"
#include <stdbool.h>
#include <stdio.h>

volatile bool spi_done = false;

void spi_callback(spi_callback_args_t * p_args)
{
    if (p_args->event == SPI_EVENT_TRANSFER_COMPLETE) spi_done = true;
}

void hal_entry(void)
{
    /* TODO: add your own code here */
    fsp_err_t err;

    err = R_SPI_Open(&g_spi1_ctrl, &g_spi1_cfg);
    assert(FSP_SUCCESS == err);

    uint8_t tx_data[4] = {0x11, 0x22, 0x33, 0x44};
    uint8_t rx_data[4] = {0};

    spi_done = false;

    err = R_SPI_WriteRead(&g_spi1_ctrl, tx_data, rx_data, sizeof(tx_data),
SPI_BIT_WIDTH_8_BITS);
    assert(FSP_SUCCESS == err);

    while (!spi_done) __NOP();

    err = R_SPI_Close(&g_spi1_ctrl);
    assert(FSP_SUCCESS == err);

    while (1)
    {
        R_BSP_SoftwareDelay(1000, BSP_DELAY_UNITS_MILLISECONDS);
    }
}

#if BSP_TZ_SECURE_BUILD
/* Enter non-secure code */
R_BSP_NonSecureEnter();
#endif

#if BSP_TZ_SECURE_BUILD

FSP_CPP_HEADER
BSP_CMSE_NONSECURE_ENTRY void template_nonsecure_callable ();

/* Trustzone Secure Projects require at least one nonsecure callable function in
order to build (Remove this if it is not required to build). */
BSP_CMSE_NONSECURE_ENTRY void template_nonsecure_callable ()
```

```

{
}
FSP_CPP_FOOTER

#endif

```

Lập trình thành ghi:

```

#include "hal_data.h"
#include "bsp_api.h"

#define BUFFER_SIZE 4
uint8_t tx_buffer[BUFFER_SIZE] = {0xAA, 0xBB, 0xCC, 0xDD};
uint8_t rx_buffer[BUFFER_SIZE] = {0};

void spi_pins_init(void) {
    R_PMISC->PWRP_b.B0WI = 0;
    R_PMISC->PWRP_b.PFSWE = 1;

    R_PFS->PORT[4].PIN[10].PmnPFS_b.PSEL = 0b00110; // SPI1_MISOB
    R_PFS->PORT[4].PIN[11].PmnPFS_b.PSEL = 0b00110; // SPI1_MOSIB
    R_PFS->PORT[4].PIN[12].PmnPFS_b.PSEL = 0b00110; // SPI1_RSPCKB

    R_PFS->PORT[4].PIN[10].PmnPFS_b.PDR = 0;
    R_PFS->PORT[4].PIN[11].PmnPFS_b.PDR = 0;
    R_PFS->PORT[4].PIN[12].PmnPFS_b.PDR = 0;

    R_PFS->PORT[4].PIN[10].PmnPFS_b.PCR = 0;
    R_PFS->PORT[4].PIN[11].PmnPFS_b.PCR = 0;
    R_PFS->PORT[4].PIN[12].PmnPFS_b.PCR = 0;

    R_PFS->PORT[4].PIN[10].PmnPFS_b.PODR = 0;
    R_PFS->PORT[4].PIN[11].PmnPFS_b.PODR = 0;
    R_PFS->PORT[4].PIN[12].PmnPFS_b.PODR = 0;

    R_PMISC->PWRP_b.PFSWE = 0;
    R_PMISC->PWRP_b.B0WI = 1;
}

void spi_init(void) {
    R_MSTP->MSTPCRB &= ~(1U << 29); // MSTPB29 = 0

    R_SPI1->SPCR = 0x00; //Clear SPCR register

    R_SPI1->SPCR_b.SPMS = 1; //Clock synchronous operation
    R_SPI1->SPCR_b.TXMD = 0; //Full-duplex
    R_SPI1->SPCR_b.MODFEN = 0; //No mode fault detect
    R_SPI1->SPCR_b.MSTR = 1; //Master mode

    R_SPI1->SPCMD[0] = 0x0700; //CPOL=0, CPHA=0, MSB first, 8-bit
    R_SPI1->SPBR = 4; //Bit rate = f(PCLKB)/(2x(n+1)x2^N) = 5Mbps (N
depend on BRDV bit)

    R_SPI1->SPCR_b.SPE = 1; // SPI enable
}

```

```

void spi_operation(void) {
    for (int i = 0; i < BUFFER_SIZE; i++) {
        /* Wait TX buffer empty */
        while (R_SPI1->SPSR_b.SPTEF == 0);

        /* Write data */
        R_SPI1->SPDR = tx_buffer[i];

        /* Wait RX completed */
        while (R_SPI1->SPSR_b.SPRF == 0);    // SPRF = 1

        /* Read data */
        rx_buffer[i] = (uint8_t)R_SPI1->SPDR;
    }
}

void hal_entry(void)
{
    /* TODO: add your own code here */
    spi_pins_init();
    spi_init();
    spi_operation();

    while (1);
}

#if BSP_TZ_SECURE_BUILD
    /* Enter non-secure code */
    R_BSP_NonSecureEnter();
#endif

}

#if BSP_TZ_SECURE_BUILD

FSP_CPP_HEADER
BSP_CMSE_NONSECURE_ENTRY void template_nonsecure_callable ();

/* Trustzone Secure Projects require at least one nonsecure callable function in
order to build (Remove this if it is not required to build). */
BSP_CMSE_NONSECURE_ENTRY void template_nonsecure_callable ()
{
}

FSP_CPP_FOOTER

#endif

```