

Universidade Federal do Rio Grande do Sul
Escola de Engenharia
Departamento de Sistemas Elétricos de Automação e Energia
ENG10032 Microcontroladores

Roteiro de Laboratório 8 ***Serial Peripheral Interface (SPI)***

Prof. Walter Fetter Lages

21 de maio de 2019

1 Objetivo

O objetivo deste laboratório é entender o funcionamento do barramento SPI e as diferenças entre uma implementação por *software* e uma implementação por *hardware*.

2 Fundamentação Teórica

O SPI é um barramento de comunicação serial síncrono utilizado para pequenas distâncias. Ele opera em *full-duplex* no modo mestre-escravo.

O SPI é baseado em quatro sinais, como mostrado na figura 1:

SCLK: *clock*, gerado pelo mestre

MOSI: Saída de dados do mestre, entrada no escravo

MISO: Entrada de dados no mestre, saída do escravo

#SS: Seleção do escravo, gerado pelo mestre

Embora menos comum, o SPI também pode funcionar em uma topologia *daisy chain*, como mostra a figura 2.

2.1 Polaridade e Fase do *Clock*

O SPI pode operar com duas polaridades e duas fases do sinal de *clock*. Normalmente estas opções são configuradas por bits com os nomes CPOL e CPHA, respectivamente.

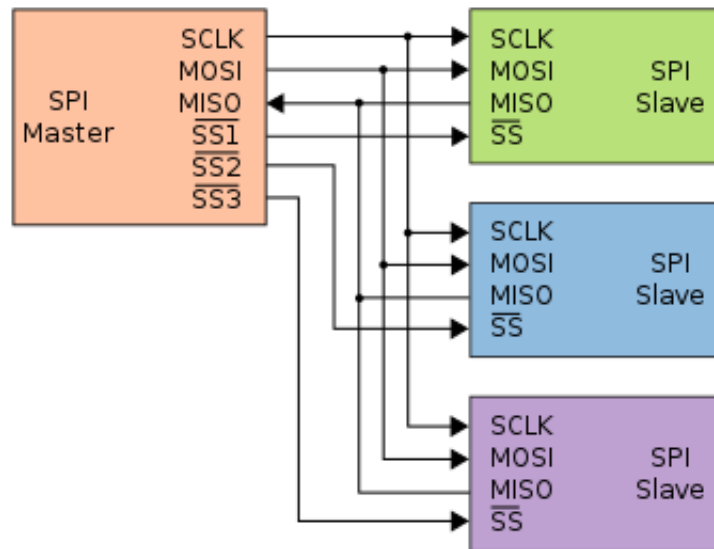


Figura 1: Barramento SPI com três escravos.

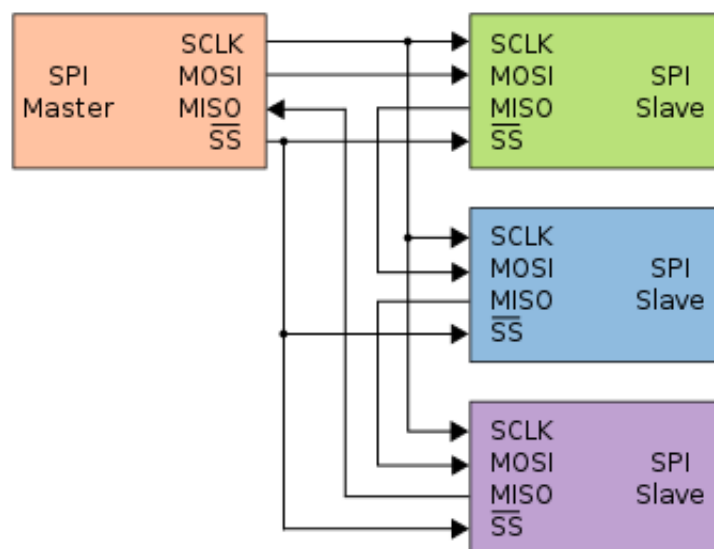


Figura 2: Barramento SPI com três escravos em *daisy chain*.

Para $CPOL=0$, o estado inativo do `clock` é em nível lógico baixo. Neste caso, quando $CPHA=0$, os dados são alterados na borda de descida do `clock` e amostrados na borda de subida do `clock`, como mostra a figura 3(a). Quando $CPHA=1$ os dados são alterados na subida do `clock` e amostrados na descida do `clock`, como mostra a figura 3(b).

Para $CPOL=1$, o estado inativo do `clock` é em nível lógico alto. Neste caso, quando $CPHA=0$, os dados são alterados na borda de subida do `clock` e amostrados na borda de descida do `clock`, como mostra a figura 3(c). Quando $CPHA=1$ os dados são alterados na descida do `clock` e amostrados na subida do `clock`, como mostra a figura 3(d).

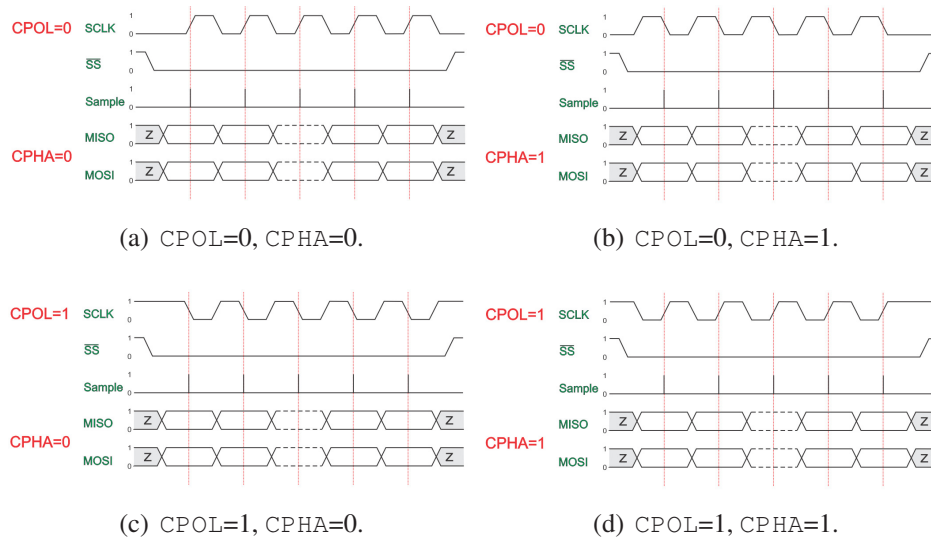


Figura 3: Polaridade e Fase do `clock`.

A tabela 1 mostra a configuração da polaridade e fase do `clock`.

Tabela 1: Configuração do `clock`.

Modo	CPOL	CPHA	Estado inativo do <code>clock</code>	Alteração dos dados	Amostragem dos dados
<code>SPI_MODE_0</code>	0	0	baixo	descida	subida
<code>SPI_MODE_1</code>	0	1	baixo	subida	descida
<code>SPI_MODE_2</code>	1	0	alto	subida	descida
<code>SPI_MODE_3</code>	1	1	alto	descida	subida

2.2 SPI na Galileo Gen 2

O protocolo SPI poderia ser implementado por *software* utilizando 4 portas GPIO quaisquer. No entanto, neste caso, a taxa máxima de transferência seria limitada pela velocidade de execução do *software*. Maiores taxas de transferência podem ser obtidas por implementações em *hardware*. O Quark X1000 possui 3 interfaces SPI. Uma é utilizada pela memória *flash*, outra é utilizada pelo conversor A/D e a terceira está disponível no conector de *shield*, conforme pinagem na tabela 2.

Tabela 2: Pinagem da interface SPI no *shield* da Galileo Gen 2.

Pino	Sinal
IO10	#SS
IO11	MOSI
IO12	MISO
IO13	SCLK

O barramento SPI disponível no conector de *shield* da Galileo Gen 2 é acessado através do dispositivo `/dev/spidev1.0`, que suporta as operações `open()`, `close()`, `read()`, `write()` e `ioctl()`. As operações de `read()` e `write()` são utilizadas para receber e transmitir dados, respectivamente, no modo *half-duplex*.

A operação `ioctl()` é usada para configurar a porta SPI e para transmitir e receber dados no modo *full-duplex*, conforme as operações mostradas na tabela 3, cujas constantes estão definidas no arquivo de cabeçalho `linux/spi/spidev.h`.

Tabela 3: Operações `ioctl()`.

Operação	Descrição
<code>SPI_IOC_RD_MODE</code>	Lê o modo do <i>clock</i>
<code>SPI_IOC_WR_MODE</code>	Escreve o modo do <i>clock</i>
<code>SPI_IOC_RD_LSB_FIRST</code>	Lê a justificação dos bits
<code>SPI_IOC_WR_LSB_FIRST</code>	Escreve a justificação dos bits
<code>SPI_IOC_RD_BITS_PER_WORD</code>	Lê o tamanho da palavra
<code>SPI_IOC_WR_BITS_PER_WORD</code>	Escreve o tamanho da palavra
<code>SPI_IOC_RD_MAX_SPEED_HZ</code>	Lê a taxa de transferência máxima
<code>SPI_IOC_WR_MAX_SPEED_HZ</code>	Escreve a taxa de transferência máxima
<code>SPI_IOC_MESSAGE(N)</code>	Envia/recebe N mensagens

A tabela 1 mostra as constantes estão definidas para configurar o modo do *clock*. As constantes `SPI_CPOL` e `SPI_CPHA` podem ser usadas como máscara para obter os bits correspondentes do modo do *clock*.

A constante `SPI_LSB_FIRST` pode ser usada para configurar a justificação dos bits.

As mensagens enviadas no modo *full-duplex* utilizam um *buffer* com o seguinte formato, definido no arquivo de cabeçalho `linux/spi/spidev.h`:

```
struct spi_ioc_transfer {
    __u64      tx_buf;
    __u64      rx_buf;

    __u32      len;
    __u32      speed_hz;

    __u16      delay_usecs;
    __u8       bits_per_word;
    __u8       cs_change;
    __u32      pad;
};
```

onde:

tx_buf: ponteiro para o *buffer* de transmissão

rx_buf: ponteiro para o *buffer* de recepção

len: tamanho dos *buffers*

speed_hz: taxa de transferência

delay_usecs: *delay* para desativar o #SS após a última transferência

bits_per_word: tamanho da palavra

cs_change: 1 para desativar o #SS antes da próxima transferência

Tipicamente, o *driver* do Linux para o controlador SPI é responsável por controlar o sinal #SS. Na Galileo Gen 2 o sinal `gpio10` (IO10) é usado como *chip select*.

No entanto, a biblioteca do Arduino tem um paradigma diferente para isso. Ao invés de assumir um *chip select* gerenciado pelo controlador SPI, o Arduino assume que qualquer saída GPIO pode ser usada como *chip select* e deixa isso para a aplicação.

Para evitar o conflito entre o Linux e o Arduino, o *driver* SPI da Galileo Gen 2 é configurado, por *default* para não usar o `gpio10` como *chip select*, mantendo a compatibilidade com o paradigma do Arduino.

No entanto, este *default* pode ser alterado para que o *driver* SPI da Galileo Gen 2 gere o *chip select* implicitamente. Para isso é necessário passar o seguinte parâmetro na linha de comando do *kernel*:

```
intel_qrk_plat_galileo_gen2.gpio_cs=1
```

Isso pode ser feito, por exemplo, adicionando-se este parâmetro na linha correspondente (a que começa com a palavra *kernel*) no arquivo `/media/card/boot/grub/grub.conf`. Note que o arquivo `grub.conf` *default* da Galileo Gen2 está configurado com duas imagens de *kernel*. A primeira é para o *kernel* armazenado na memória *flash* e a segunda é para o *kernel* armazenado no cartão microSD e é esta que deve ser alterada.

Pode-se verificar que a `gpio10` foi reservada para uso do *driver* SPI através do comando:

```
echo 10 > /sys/class/gpio/export
```

que se a porta está reservada para o *driver* SPI, retornará erro pois neste caso ela não pode ser exportada para ser usada como GPIO.

A listagem 1 é um programa para mostrar a configuração da interface SPI, ou seja, modo do *clock*, justificação dos bits, tamanho da palavra e taxa de transferência máxima.

Listagem 1: Programa para mostrar a configuração da interface SPI.

```
1 #include <fcntl.h>
2 #include <stdint.h>
3 #include <stdio.h>
4 #include <unistd.h>
5 #include <sys/ioctl.h>
6 #include <linux/spi/spidev.h>
7
8 int main(int argc, char *argv[])
9 {
10     int fd;
11     uint8_t mode;
12     uint8_t lsb;
13     uint8_t bpw;
14     uint32_t rate;
15
16     if(argc != 2)
17     {
```

```

18         printf("Usage: spistat device\n");
19         return -1;
20     }
21     if((fd=open(argv[1],O_RDWR))== -1)
22     {
23         perror("Can't open device");
24         return -1;
25     }
26     if(ioctl(fd,SPI_IOC_RD_MODE,&mode))
27     {
28         perror("Can't read clock mode");
29         return -1;
30     }
31     printf("Clock mode: 0x%x CPOL=%d CPHA=%d\n",
32           mode,
33           (mode & SPI_CPOL)? 1:0,
34           (mode & SPI_CPHA)? 1:0);
35     if(ioctl(fd,SPI_IOC_RD_LSB_FIRST,&lsb))
36     {
37         perror("Can't read LSB mode");
38         return -1;
39     }
40     printf("LSB mode: 0x%x\n",lsb);
41     if(ioctl(fd,SPI_IOC_RD_BITS_PER_WORD,&bpw))
42     {
43         perror("Can't read bits per word");
44         return -1;
45     }
46     printf("Bits per word: %d\n",bpw);
47     if(ioctl(fd,SPI_IOC_RD_MAX_SPEED_HZ,&rate))
48     {
49         perror("Can't read maximal rate");
50         return -1;
51     }
52     printf("Maximal transfer rate: %d Hz\n",rate);
53     close(fd);
54     return 0;
55 }

```

3 Experimentos

1. Modifique a linha de comando do *kernel* da Galileo para que o sinal #SS do barramento SPI seja gerado automaticamente. Atente para alterar a linha de comando do *kernel* que está no microSD e não na *flash*.
2. Reinicialize a Galileo e verifique se a configuração do sinal #SS está correta.
3. Crie o grupo `spi` e inclua o seu usuário nele.
4. Baixe do Moodle <<http://moodle.ece.ufrgs.br>> o *script* de inicialização para configurar a Galileo para uso do barramento SPI e as permissões de leitura e escrita para o grupo `spi` no dispositivo `/dev/spidev1.0`.
5. Instale o *script* de inicialização e reinicialize a Galileo.
6. Compile e execute o programa mostrado na Listagem 1.
7. Faça um programa para transmitir dados utilizando a interface SPI da Galileo Gen2.
8. Baixe do Moodle <<http://moodle.ece.ufrgs.br>> o programa que implementa em *software* um escravo SPI. Este escravo usa os pinos IO4 como SCLK, IO5 como MOSI, IO6 como MISO e IO9 como #SS.
9. Procure entender em linhas gerais o código fonte do programa e gere o executável.
10. Baixe do Moodle <<http://moodle.ece.ufrgs.br>> o *script* de inicialização para configurar os pinos IO4, IO5, IO6 e IO9 para funcionamento do escravo SPI por *software*.
11. Conecte os pinos da interface SPI da Galileo Gen 2 nos pinos utilizados pelo programa que implementa o escravo SPI.
12. Teste os programas utilizando uma taxa de transferência de 1 kHz. Execute o mestre SPI em uma janela e o escravo em outra.