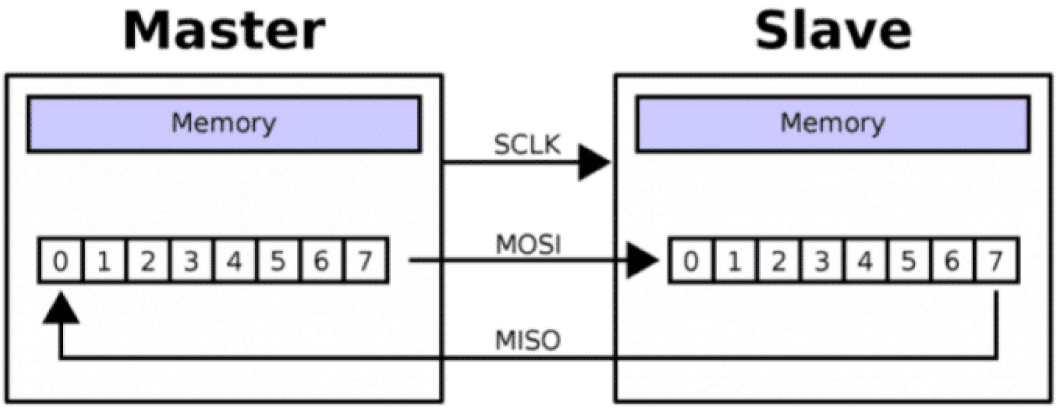
Comunicação SPI em Linux

Por **Vinicius Maciel** - 07/04/2015



ÍNDICE DE CONTEÚDO [MOSTRAR]

Este post faz parte da série Device Drivers para Linux Embarcado. Leia também os outros posts da série:

- Device Drivers para Linux Embarcado - Introdução
- Linux Device Drivers - Diferenças entre Drivers de Plataforma e de Dispositivo
- Exemplo de driver para Linux Embarcado
- Comunicação SPI em Linux
- Exemplo de Device Driver I2C para Linux Embarcado

O protocolo SPI

Dando continuidade aos artigos sobre acesso a dispositivos típicos de sistemas embarcados no Linux, vamos abordar neste artigo **Comunicação SPI em Linux**. Por enquanto apenas visando softwares em espaço de usuário.

O protocolo SPI (*Serial Peripheral Interface*) foi definido pela Motorola. É um protocolo serial síncrono semelhante ao protocolo I2C, mas que utiliza três fios para comunicação e pode alcançar velocidades superiores ao I2C. A interface física é composta pelos sinais MOSI, MISO e SCLK que ligam a todos os dispositivos na forma de barramento. Pode existir também um quarto fio para seleção do dispositivo com o qual a comunicação será feita.

Assim como o I2C, o SPI apresenta o conceito de mestre e escravo da comunicação. O mestre inicia a transferência de dados e controla o sinal de clock para estabelecer o sincronismo. A transferência de dados processa-se em *full-duplex* sobre os sinais MOSI (*Master Output Slave Input*) e MISO (*Master Input Slave Output*). A taxa de transferência é definida pelo processador, no papel de mestre, através do sinal SCLK. A seleção do periférico é feita através do sinal SSn. Nessa configuração, pode haver um mestre e vários escravos no barramento SPI, como mostrado na Figura 1.

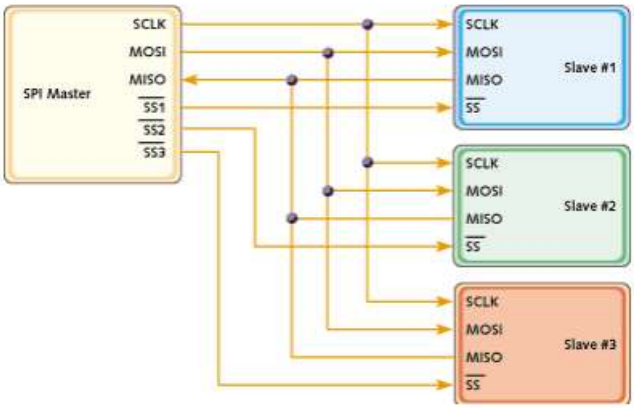


Figura 1 - Ligação entre um mestre e escravos no barramento SPI.

O fluxo de dados é controlado pelo mestre através do sinal de clock SCLK. Só há transferência enquanto o mestre pulsar o sinal SCLK. Em repouso o sinal SCLK encontra-se estável com o valor lógico definido por CPOL.

Habilitando o driver SPI em Linux

Assim como para o I2C, o Linux disponibiliza um driver genérico para SPI. Ele cria entradas no diretório */dev* para que o usuário possa acessar o dispositivo SPI como um arquivo através da funções *open()*, *close()*, *read()* e *write()*. Por ser um driver genérico, ele pode não atender a todos os requisitos do seu projeto. Se você pretende usar o chipset ENC28J60 para construir uma interface ethernet, por exemplo, você precisará de um driver específico. Para esse chipset já existe um driver no Linux. Mas, para um outro chipset que não tenha suporte no Linux, você deverá desenvolvê-lo.

Saiba que, para se comunicar com um dispositivo SPI em Linux, por meio da sua infraestrutura, o processador da placa que você está usando precisa ter o driver do controlador SPI presente no kernel. No artigo Linux Device Drivers – Diferenças entre Drivers de Plataforma e de Dispositivo explico em detalhes o funcionamento geral dos drivers. Consulte essa referência!

Agora vamos habilitar o driver SPI no kernel. Selecione *Device Driver*:

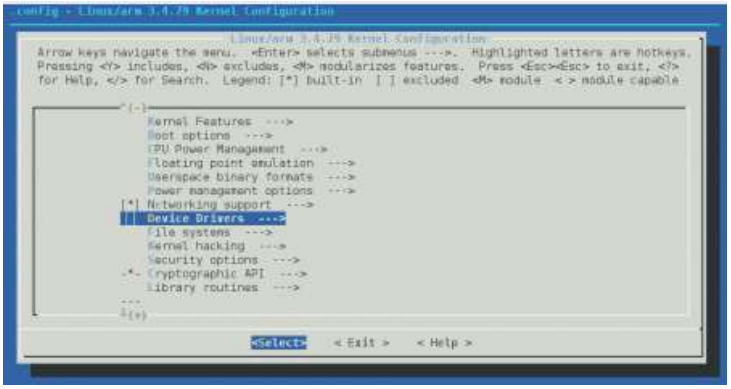


Figura 2 - Habilitando o driver de SPI (menu Device Drivers).

Habilite o suporte a SPI apertando ‘y’, depois aperte ENTER:

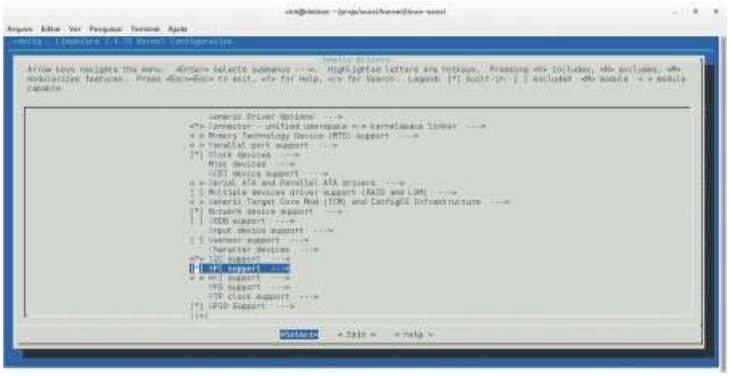


Figura 3 - Habilitando o driver de SPI (item SPI support).

Habilite o suporte ao driver SPI em Linux como módulo apertando ‘m’ ou ‘y’ para deixar o driver embutido no kernel:

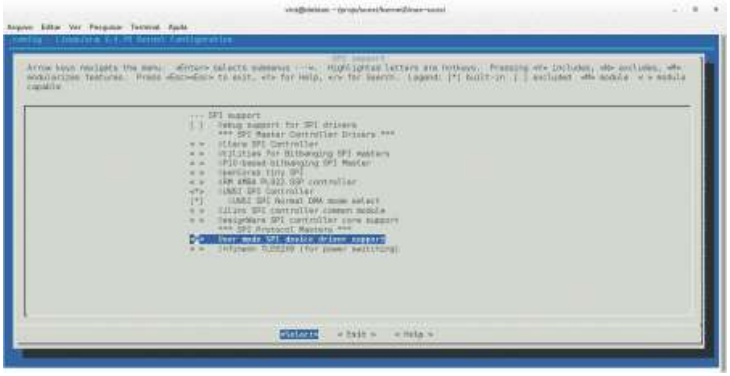


Figura 4 - Habilitando o driver de SPI (módulo ou built-in).

Agora basta compilar o kernel!

Comunicação SPI em Linux

Agora veremos um exemplo de software para escrever e ler um dispositivo qualquer que utiliza o protocolo SPI. O dispositivo mais simples para fazer um teste é uma memória Flash. Mas pode ser usado qualquer outro dispositivo, como um módulo RFID.

Exemplo de software para SPI:

```
/*
 * Exemplo de software para comunicacao SPI
 *
 * Autor: Vinicius Maciel
 *
 * comando para compilacao: gcc spi_teste.c -o spi_teste
 *
 * comando para cross compilacao: arm-linux-gnueabihf-gcc spi_teste.c -o spi_teste
 */

#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/types.h>

#include "spidev.h"

#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))

/* Funcao que imprime mensagem de erro e aborta o programa
 *
 */
static void pabort(const char *s)
{
```

```
29      perror(s);
30      abort();
31  }
32
33  // Nome do dispositivo de driver SPI no diretorio /dev
34  static const char *device = "/dev/spidev0.0"; // (Mude de acordo com sua placa)
35  // Modo de operacao do controlador SPI do SoC
36  static uint32_t mode;
37  // Quantidade de bits da palavra de transferencia SPI
38  static uint8_t bits = 8;
39  // Velocidade de comunicao
40  static uint32_t speed = 1000000;
41  // Delay entre bytes transferidos, caso necessario
42  static uint16_t delay;
43
44  /*
45   * Funcao que realiza uma transferencia full duplex,
46   * ou seja, que escreve e ler ao mesmo tempo
47   *
48   * Parametros ----
49   * fd: inteiro retornado pela funcao open()
50   * tx: para de escrita para mensagem SPI
51   * rx: buffer de leitura de mensagem SPI
52   */
53  static void transfer(int fd, uint8_t *tx, uint8_t *rx)
54  {
55      int ret;
56
57      // Estrutura que contem as informacoes para a transmissao da mensagem
58      struct spi_ioc_transfer tr = {
59       .tx_buf = (unsigned long)tx,
60       .rx_buf = (unsigned long)rx,
61       .len = ARRAY_SIZE(tx),
62       .delay_usecs = delay,
63       .speed_hz = speed,
64       .bits_per_word = bits,
65      };
66
67      // Funcao que faz a transferencia full duplex
68      ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
69      if (ret < 1)
70       pabort("Nao foi possivel enviar mensagem SPI");
71
72      // Imprimi a mensagem recebida, caso haja alguma
73      for (ret = 0; ret < ARRAY_SIZE(rx); ret++) {
74       printf("%x ", rx[ret]);
75      }
76      puts("");
77  }
78
79
80  int main(int argc, char *argv[])
81  {
82      int ret = 0, i;
83      int fd;
84      // Buffers para leitura e escrita da mensagem SPI
85      uint8_t tx_buffer[32], rx_buffer[32];
86
87
88      // Abri o dispositivo de driver SPI e retorna um inteiro
89      fd = open(device, O_RDWR);
90      if (fd < 0)
91   pabort("Erro ao abrir o dispositivo");
92
93      // Escolha outros modos de operacao que o SPI da sua placa suporta
94      mode = SPI_CPHA | SPI_CPOL | SPI_MODE_0;
95
96      // Configura o modo de operacao
97      ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);
98      if (ret == -1)
99       pabort("Erro ao setar o modo do SPI");
100
101     ret = ioctl(fd, SPI_IOC_RD_MODE32, &mode);
102     if (ret == -1)
103      pabort("Erro ao setar o modo do SPI");
104
105     // Configura o tamanho da palavra de transferencia SPI para escrita
106     ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
107     if (ret == -1)
108      pabort("Erro ao setar os bits por palavra");
109
110     // Configura o tamanho da palavra de transferencia SPI para leitura
111     ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
112     if (ret == -1)
113      pabort("Erro ao ler os bits por palavra");
114
115     // Configura a maxima velocidade de transferencia para escrita
```

```
    ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
    if (ret == -1)
     pabort("Erro ao setar a velocidade maxima em HZ");

    // Configura a maxima velocidade de transferencia para leitura
    ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
    if (ret == -1)
     pabort("Erro ao ler a velocidade maxima em HZ");

    // Imprimi informacoes de configuracao do SPI
    printf("Modo SPI: 0x%x\n", mode);
    printf("bits por palavra: %d\n", bits);
    printf("Maxima velocidade: %d Hz (%d KHz)\n", speed, speed/1000);

    // Preenche o buffer de transferencia com numeros de 0 a 9
    for (i = 0; i < 9; i++)
 tx_buffer[i] = i;

    // Faz a transferencia full duplex
    transfer(fd, tx_buffer, rx_buffer);

    // Fecha o dispositivo de driver SPI
    close(fd);

    return ret;
}
```

Para que o código seja compilado, é necessário incluir o seguinte código de cabeçalho:

```
/*
 * include/linux/spi/spidev.h
 *
 * Copyright (C) 2006 SWAPP
 *  Andrea Paterniani <a.paterniani@swapp-eng.it>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#ifndef SPIDEV_H
#define SPIDEV_H

#include <linux/types.h>

/* User space versions of kernel symbols for SPI clocking modes,
 * matching <linux/spi/spi.h>
 */

#define SPI_CPHA 0x01
#define SPI_CPOL 0x02

#define SPI_MODE_0 (0|0)
#define SPI_MODE_1 (0|SPI_CPHA)
#define SPI_MODE_2 (SPI_CPOL|0)
#define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)

#define SPI_CS_HIGH 0x04
#define SPI_LSB_FIRST 0x08
#define SPI_3WIRE 0x10
#define SPI_LOOP 0x20
#define SPI_NO_CS 0x40
#define SPI_READY 0x80
#define SPI_TX_DUAL 0x100
#define SPI_TX_QUAD 0x200
#define SPI_RX_DUAL 0x400
#define SPI_RX_QUAD 0x800
```

```
49
50  /*---------------------------------------------------------------------------*/
51
52  /* IOCTL commands */
53
54  #define SPI_IOC_MAGIC 'k'
55
56  /**
57   * struct spi_ioc_transfer - describes a single SPI transfer
58   * @tx_buf: Holds pointer to userspace buffer with transmit data, or null.
59   *  If no data is provided, zeroes are shifted out.
60   * @rx_buf: Holds pointer to userspace buffer for receive data, or null.
61   * @len: Length of tx and rx buffers, in bytes.
62   * @speed_hz: Temporary override of the device's bitrate.
63   * @bits_per_word: Temporary override of the device's wordsize.
64   * @delay_usecs: If nonzero, how long to delay after the last bit transfer
65   *  before optionally deselecting the device before the next transfer.
66   * @cs_change: True to deselect device before starting the next transfer.
67   *
68   * This structure is mapped directly to the kernel spi_transfer structure;
69   * the fields have the same meanings, except of course that the pointers
70   * are in a different address space (and may be of different sizes in some
71   * cases, such as 32-bit i386 userspace over a 64-bit x86_64 kernel).
72   * Zero-initialize the structure, including currently unused fields, to
73   * accommodate potential future updates.
74   *
75   * SPI_IOC_MESSAGE gives userspace the equivalent of kernel spi_sync().
76   * Pass it an array of related transfers, they'll execute together.
77   * Each transfer may be half duplex (either direction) or full duplex.
78   *
79   *  struct spi_ioc_transfer mesg[4];
80   *  ...
81   *  status = ioctl(fd, SPI_IOC_MESSAGE(4), mesg);
82   *
83   * So for example one transfer might send a nine bit command (right aligned
84   * in a 16-bit word), the next could read a block of 8-bit data before
85   * terminating that command by temporarily deselecting the chip; the next
86   * could send a different nine bit command (re-selecting the chip), and the
87   * last transfer might write some register values.
88   */
89  struct spi_ioc_transfer {
90   __u64 tx_buf;
91   __u64 rx_buf;
92
93   __u32 len;
94   __u32 speed_hz;
95
96   __u16 delay_usecs;
97   __u8 bits_per_word;
98   __u8 cs_change;
99   __u8 tx_nbits;
100  __u8 rx_nbits;
101  __u16 pad;
102
103  /* If the contents of 'struct spi_ioc_transfer' ever change
104   * incompatibly, then the ioctl number (currently 0) must change;
105   * ioctls with constant size fields get a bit more in the way of
106   * error checking than ones (like this) where that field varies.
107   *
108   * NOTE: struct layout is the same in 64bit and 32bit userspace.
109   */
110 };
111
112 /* not all platforms use <asm-generic/ioctl.h> or _IOC_TYPECHECK() ... */
113 #define SPI_MSGSIZE(N) \
114  ((((N)*(sizeof (struct spi_ioc_transfer))) < (1 << _IOC_SIZEBITS)) \
115  ? ((N)*(sizeof (struct spi_ioc_transfer))) : 0)
116 #define SPI_IOC_MESSAGE(N) _IOW(SPI_IOC_MAGIC, 0, char[SPI_MSGSIZE(N)])
117
118
119 /* Read / Write of SPI mode (SPI_MODE_0..SPI_MODE_3) (limited to 8 bits) */
120 #define SPI_IOC_RD_MODE _IOR(SPI_IOC_MAGIC, 1, __u8)
121 #define SPI_IOC_WR_MODE _IOW(SPI_IOC_MAGIC, 1, __u8)
122
123 /* Read / Write SPI bit justification */
124 #define SPI_IOC_RD_LSB_FIRST _IOR(SPI_IOC_MAGIC, 2, __u8)
125 #define SPI_IOC_WR_LSB_FIRST _IOW(SPI_IOC_MAGIC, 2, __u8)
126
127 /* Read / Write SPI device word length (1..N) */
128 #define SPI_IOC_RD_BITS_PER_WORD    _IOR(SPI_IOC_MAGIC, 3, __u8)
129 #define SPI_IOC_WR_BITS_PER_WORD    _IOW(SPI_IOC_MAGIC, 3, __u8)
130
131 /* Read / Write SPI device default max speed hz */
132 #define SPI_IOC_RD_MAX_SPEED_HZ _IOR(SPI_IOC_MAGIC, 4, __u32)
133 #define SPI_IOC_WR_MAX_SPEED_HZ _IOW(SPI_IOC_MAGIC, 4, __u32)
134
135 /* Read / Write of the SPI mode field */
```

```
136 #define SPI_IOC_RD_MODE32 _IOR(SPI_IOC_MAGIC, 5, __u32)
137 #define SPI_IOC_WR_MODE32 _IOW(SPI_IOC_MAGIC, 5, __u32)
138
139 #endif /* SPIDEV_H */
```

Este artigo foi originalmente publicado no site Software Livre.

Saiba mais sobre SPI

- Comunicação SPI

- Linux Kernel Documentation

- Gerando PWM com a Raspberry PI

Outros artigos da série

<< Exemplo de driver para Linux EmbarcadoExemplo de Device Driver I2C para Linux Embarcado >>

Este post faz da série Device Drivers para Linux Embarcado. Leia também os outros posts da série:

- Device Drivers para Linux Embarcado - Introdução
- Linux Device Drivers - Diferenças entre Drivers de Plataforma e de Dispositivo
- Exemplo de driver para Linux Embarcado
- Comunicação SPI em Linux
- Exemplo de Device Driver I2C para Linux Embarcado

Vinicius Maciel

http://www.softwarelivre.blog.br/

Cursando Tecnologia em Telemática no IFCE. Trabalho com programação de Sistemas Embarcados desde 2009. Tenho experiência em desenvolvimento de software para Linux embarcado em C/C++. Criação ou modificação de devices drivers para o sistema operacional Linux. Uso de ferramentas open source para desenvolvimento e debug incluindo gcc, gdb, eclipse.

G+