SPI Device Drivers

Mg. Ing. Pablo Slavkin
Mg. Ing. Hanes N. Sciarrone
MSE - 2024

Implementación de Manejadores de Dispositivos

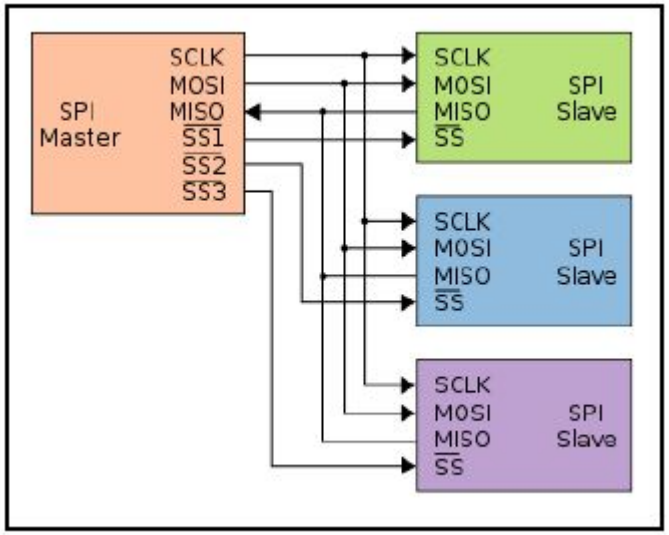
SPI Device Drivers

- Introducción
- Arquitectura del driver
- Accediendo al cliente

Introducción

Introducción

- El bus SPI es uno muy bien conocido, simple en su conexión y en la comprensión de su funcionamiento.
- Consta de (por lo menos) 4 líneas:
  - MISO (Master Input Slave Output)
  - MOSI (Master Output Slave Input)
  - SCK (Serial Clock)
  - CS (Chip Select)
- Alcanza velocidades de 80 MHz aunque no tiene límite teórico.



Arquitectura del driver

Arquitectura del driver

- El header requerido para utilizar las estructuras correspondientes a SPI es **linux/spi/spi.h**

- Un dispositivo SPI se representa en el kernel como una instancia de una estructura **spi_device**.

- La instancia del driver que administra estos dispositivos es la estructura **spi_driver**.

Arquitectura del driver

- La estructura **spi_device,** que representa el dispositivo en el kernel, tiene los siguientes campos (algunos no mostrados):

```
struct  spi_device {
    struct device dev;
    struct spi_master *master;
    u32 max_speed_hz;
    u8 chip_select;
    u8 bits_per_word;
    u16 mode;
    int cs_gpio;
};
```

Arquitectura del driver

- **master:** Representa el controlador SPI (del bus) donde el dispositivo esta conectado
- **max_speed_hz:** Maxima frecuencia de clock a utilizar. Este parametro puede ser cambiado en el driver, incluso entre trasferencias.
- **chip_select:** Representa el pin de CS que se utiliza para el dispositivo.
- **mode:** Define el modo de SPI (CPOL y CPHA). Los datos son transmitidos MSB primero, pero se puede configurar.

Arquitectura del driver

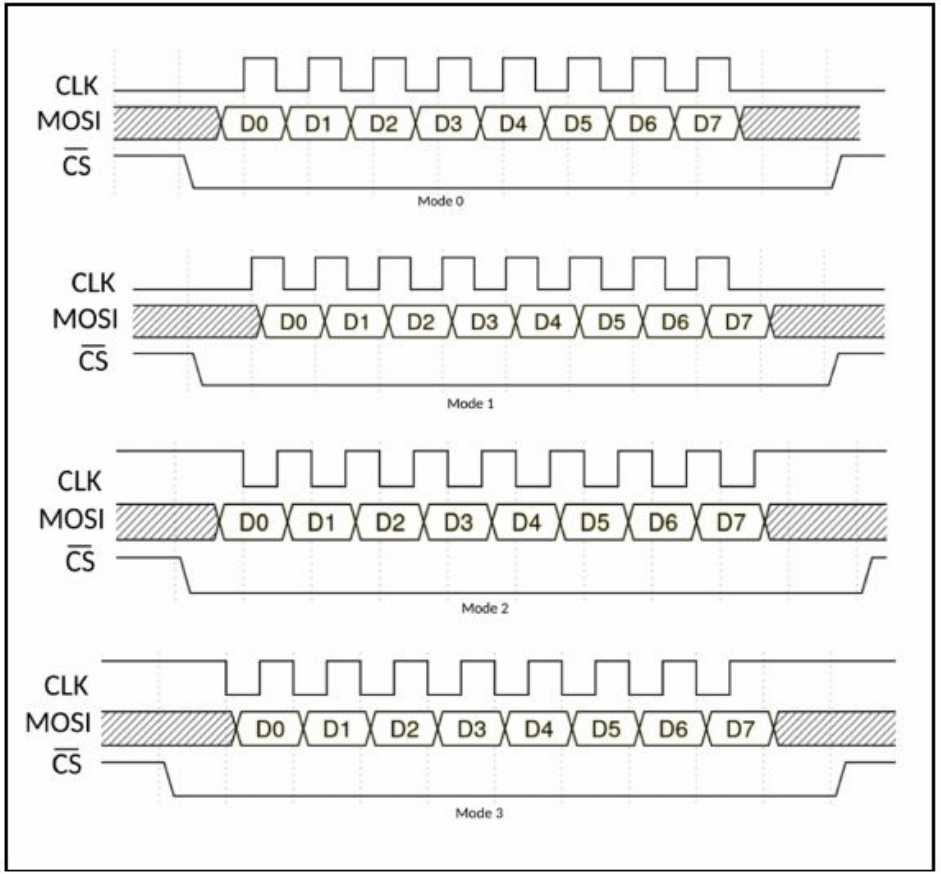
- **mode (cont):** Se utilizan dos macros para definir CPOL y CPHA:
  - CPOL (Polaridad de clock): 0 para nivel LOW de reposo y primer flanco ascendente, 1 para nivel HIGH de reposo y primer flanco descendente.
  - CPHA (Fase de clock): 0 para hacer latch de los datos durante los flancos descendentes, 1 para hacerlo durante los flancos ascendentes.
- Dentro del archivo /include/linux/spi/spi.h se encuentran dos macros para esto:

```
#define SPI_CPHA 0x01
#define SPI_CPOL 0x02
```

Arquitectura del driver

Mode	CPOL	CPHA	Kernel macro
0	0	0	`#define SPI_MODE_0 (0|0)`
1	0	1	`#define SPI_MODE_1 (0|SPI_CPHA)`
2	1	0	`#define SPI_MODE_2 (SPI_CPOL|0)`
3	1	1	`#define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)`

Arquitectura del driver



Arquitectura del driver

- En el caso de la estructura **spi_driver**, esta es muy similar a la estructrura correspondiente para I2C.
- Representa el driver que administra los dispositivos representados con **spi_device**.

```
struct spi_driver {
    const struct spi_device_id *id_table;
    int (*probe)(struct spi_device *spi);
    int (*remove)(struct spi_device *spi);
    void (*shutdown)(struct spi_device *spi);
    struct device_driver driver;
};
```

Arquitectura del driver

- La función **probe()** es responsable de iniciar el dispositivo y registrarlo en el framework apropiado.
- Como se hizo anteriormente, utilizaremos el framework misc.
- La función **probe()** recibe un solo argumento: Un puntero a **struct spi_device** que representa el dispositivo en sí.
- Este argumento es pasado por el kernel al momento de invocar a **probe()** luego del matching loop.

Arquitectura del driver

- En la funcion probe() puede llevarse registro de datos privados (de cada dispositivo).
- Esto es virtud de un puntero void dentro de la estructura **spi_device.**
- Para esto se utilizan las funciones:

```
/* set the data */
void spi_set_drvdata(struct *spi_device, void *data);
/* Get the data back */
void *spi_get_drvdata(const struct *spi_device);
```

Arquitectura del driver

- La función **remove()** es responsable de apagar el dispositivo y anular el registro en el framework apropiado.

- Recibe un solo argumento:
  - Un puntero a **struct spi_device** que representa el dispositivo en sí. Es el mismo que se pasa a la función **probe()**.

- De esta manera, la inicialización del dispositivo se hace dentro de la función **probe()** y el apagado en la función **remove()**

Arquitectura del driver

- La inicialización del driver y su registro en el kernel son muy similares al caso de un driver I2C.
- Para evitar el boilerplate, se utiliza la macro **module_spi_driver()**.
- Esta, como en el caso anterior, internamente llama a **spi_register_driver()** y **spi_unregister_driver()**.
- Además define los puntos de entrada y salida del módulo (*init* y *exit*).

Arquitectura del driver

- Al igual que para I2C, se puede aprovisionar de dispositivos mediante una tabla de ids.
- Esta tabla contiene instancias de estructuras **spi_device_id**.
- Para mantener la línea con lo expuesto en la clase anterior, solamente veremos el método OF.
- En el DT, el campo <reg> que en I2C representaba la dirección del dispositivo en el bus, ahora representa la línea CS.
- En concreto, es un índice para recorrer una lista de CS disponibles.

Arquitectura del driver

```
ecspi1 {
    fsl,spi-num-chipselects = <3>;
    cs-gpios = <&gpio5 17 0>, <&gpio5 17 0>, <&gpio5 17 0>;
    pinctrl-0 = <&pinctrl_ecspi1 &pinctrl_ecspi1_cs>;
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx6q-ecspi", "fsl,imx51-ecspi";
    reg = <0x02008000 0x4000>;
    status = "okay";
};
```

Arquitectura del driver

```
&ecspi1 {
    ad7606r8_0: ad7606r8@0 {
    compatible = "ad7606-8";
    reg = <0>;
    spi-max-frequency = <1000000>;
    interrupt-parent = <&gpio4>;
    interrupts = <30 0x0>;
};
```

Arquitectura del driver

- Como en el caso de I2C, podemos observar una propiedad llamada **spi-max-frequency.**
- En el momento que se acceda al dispositivo, el driver controlador de bus asegura que no se exceda esa frecuencia.
- Otras propiedades comunes de utilizar son:
  - **spi-pol**: Es una propiedad booleana. Estando presente indica que se requiere polaridad inversa.
  - **spi-cpha**: Propiedad booleana, estando presente indica CPHA=1.
  - **spi-cs-high**: Propiedad booleana. Invierte el nivel de CS.

Arquitectura del driver

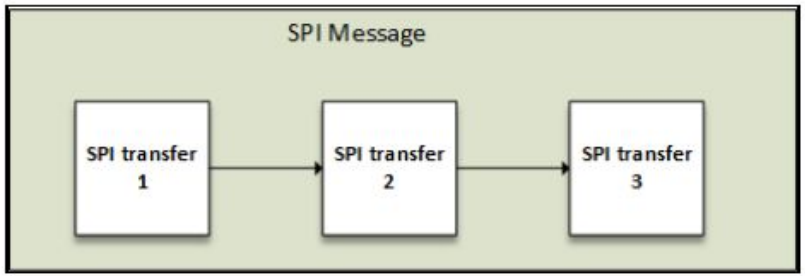
- Nuevamente, la exposición al driver del DT se hace mediante la macro **MODULE_DEVICE_TABLE** y una tabla de **of_device_id**.
- Este punto es exactamente igual al método utilizado para I2C.

```
static struct spi_driver foo_driver = {
    .driver = {
    .name = "foo",
    .of_match_table = of_match_ptr(foobar_of_match),
    },
    .probe = my_spi_probe,
};
```

Accediendo al cliente

Accediendo al cliente

- El modelo de I/O para SPI consiste en un set de mensajes dentro de una queue.
- En esta queue se ingresan estructuras **spi_message** que son procesados sincrónica o asincrónicamente.
- Un mensaje consiste en una o más estructuras **spi_transfer**.
- Cada estructura **spi_transfer** representa una transferencia full duplex.



Accediendo al cliente

- La estructura **spi_transfer** posee los siguientes campos:
  - **tx_buf**: Este buffer contiene los datos a ser escritos. Si es NULL la transacción será de solo lectura.
  - **rx_buff**: Buffer que almacena la información leída. NULL para una transacción de solo lectura.
  - **len**: La longitud de los buffers anteriores en bytes. Implica que ambos buffers deben ser del mismo tamaño.
  - **speed_hz**: Sobre escribe la velocidad por defecto, pero solo para esta transferencia. Para usar el valor por defecto se le asigna 0x00.

Accediendo al cliente

- La estructura spi_trasnsfer posee los siguientes campos (cont.):
  - **bits_per_word**: Cantidad de bits por palabra. Seteado a 0x00 utiliza el tamaño por defecto (seteado con anterioridad).
  - **cs_change**: Determina el estado de CS luego de que termina la transferencia.
  - **delay_usecs**: Representa el delay en us despues de esta transferencia antes de que se cambie el estado de CS.

Accediendo al cliente

- Por otro lado, la estructura **spi_message** es utilizada para automaticamente envolver una o mas transacciones.
- El bus SPI es retenido por el driver hasta que todas las transferencias que constituyen **spi_message** culminan.
- La estructura **spi_message** contiene los siguientes campos:
  - **transfers**: Lista de transferencias (list_head).
  - **complete**: Callback que se invoca cuando todas las transacciones terminan, se le pasa como argumento un puntero void (context).

Accediendo al cliente

- La estructura **spi_message** contiene los siguientes campos (cont.):
  - **frame_lentgth**: Se setea automáticamente con la cantidad de bytes que componen el mensaje.
  - **actual_length**: El numero de bytes transferidos en todos los segmentos exitosos.
  - **status**: Muestra el estado de las transferencias. 0x00 en éxito, **errno** en caso contrario.

Accediendo al cliente

- **NOTA**: Los elementos **spi_transfer** en un mensaje son procesados de manera FIFO.
- No se debe acceder al buffer de transferencia para evitar corrupción de datos.
- Antes de poder ingresar un mensaje al bus, se debe inicializar con la función **spi_message_init()**.
- Esta función inicializa la estructura **spi_message** y principalmente la lista de **spi_transfer** en ella.

Accediendo al cliente

- Para cada elemento que se desee incorporar al mensaje, se debe utilizar la funcion **spi_message_add_tail()**.
- Esta funcion recibe dos argumentos:
  - La estructura mensaje a la cual se desea agregar una transferencia
  - La transferencia en sí (**spi_transfer**).
- Para iniciar la transferencia se utilizan las funciones:
  - int spi_sync(struct spi_device *spi, struct spi_message *message)
  - int spi_async(struct spi_device *spi, struct spi_message *message)

Accediendo al cliente

- Existen algunas helper functions construidas en base a spi_sync():
  - int spi_read(struct spi_device *spi, void *buf, size_t len)
  - int spi_write(struct spi_device *spi, const void *buf, size_t len)
  - int spi_write_then_read()
- La lista completa se encuentra en **include/linux/spi/spi.h**

Gracias.