# 一、SPI驱动框架

## 1.1 裸机下的SPI驱动框架

 首先编写SPI控制器驱动，bsp\_spi.c和bsp\_spi.h。最终提供SPI读写函数：spich0\_readwrite\_byte

 spi具体芯片驱动：ICM20608

## 1.2、Linux下的SPI驱动框架

主机控制器驱动：SOC的SPI外设驱动，此驱动是半导体原厂编写好的，为spi-imx.c，当spi控制器的设备和驱动匹配以后，spi\_imx\_probe函数就会执行，完善I.MX6ULL的SPI控制器驱动。

SPI控制器驱动核心就是spi\_master的构建，spi\_master里面就有如何通过SPI控制器与SPI外设进行通信的函数，此函数是原厂编写的。

spi\_master->transfer

spi\_master-> transfer\_one\_message 6ULL主机控制器使用此函数，

设备驱动：具体的SPI芯片驱动

## 1.3、SPI主机驱动

###  1、spi\_master

struct spi\_master {

 struct device dev;

 struct list\_head list;

 /\* other than negative (== assign one dynamically), bus\_num is fully

 \* board-specific. usually that simplifies to being SOC-specific.

 \* example: one SOC has three SPI controllers, numbered 0..2,

 \* and one board's schematics might show it using SPI-2. software

 \* would normally use bus\_num=2 for that controller.

 \*/

 s16 bus\_num;

 /\* chipselects will be integral to many controllers; some others

 \* might use board-specific GPIOs.

 \*/

 u16 num\_chipselect;

 /\* some SPI controllers pose alignment requirements on DMAable

 \* buffers; let protocol drivers know about these requirements.

 \*/

 u16 dma\_alignment;

 /\* spi\_device.mode flags understood by this controller driver \*/

 u16 mode\_bits;

 /\* bitmask of supported bits\_per\_word for transfers \*/

 u32 bits\_per\_word\_mask;

#define SPI\_BPW\_MASK(bits) BIT((bits) - 1)

#define SPI\_BIT\_MASK(bits) (((bits) == 32) ? ~0U : (BIT(bits) - 1))

#define SPI\_BPW\_RANGE\_MASK(min, max) (SPI\_BIT\_MASK(max) - SPI\_BIT\_MASK(min - 1))

 /\* limits on transfer speed \*/

 u32 min\_speed\_hz;

 u32 max\_speed\_hz;

 /\* other constraints relevant to this driver \*/

 u16 flags;

#define SPI\_MASTER\_HALF\_DUPLEX BIT(0) /\* can't do full duplex \*/

#define SPI\_MASTER\_NO\_RX BIT(1) /\* can't do buffer read \*/

#define SPI\_MASTER\_NO\_TX BIT(2) /\* can't do buffer write \*/

#define SPI\_MASTER\_MUST\_RX BIT(3) /\* requires rx \*/

#define SPI\_MASTER\_MUST\_TX BIT(4) /\* requires tx \*/

 /\* lock and mutex for SPI bus locking \*/

 spinlock\_t bus\_lock\_spinlock;

 struct mutex bus\_lock\_mutex;

 /\* flag indicating that the SPI bus is locked for exclusive use \*/

 bool bus\_lock\_flag;

 /\* Setup mode and clock, etc (spi driver may call many times).

 \*

 \* IMPORTANT: this may be called when transfers to another

 \* device are active. DO NOT UPDATE SHARED REGISTERS in ways

 \* which could break those transfers.

 \*/

 int (\*setup)(struct spi\_device \*spi);

 /\* bidirectional bulk transfers

 \*

 \* + The transfer() method may not sleep; its main role is

 \* just to add the message to the queue.

 \* + For now there's no remove-from-queue operation, or

 \* any other request management

 \* + To a given spi\_device, message queueing is pure fifo

 \*

 \* + The master's main job is to process its message queue,

 \* selecting a chip then transferring data

 \* + If there are multiple spi\_device children, the i/o queue

 \* arbitration algorithm is unspecified (round robin, fifo,

 \* priority, reservations, preemption, etc)

 \*

 \* + Chipselect stays active during the entire message

 \* (unless modified by spi\_transfer.cs\_change != 0).

 \* + The message transfers use clock and SPI mode parameters

 \* previously established by setup() for this device

 **\*/**

 **int (\*transfer)(struct spi\_device \*spi,**

 **struct spi\_message \*mesg);**

 /\* called on release() to free memory provided by spi\_master \*/

 void (\*cleanup)(struct spi\_device \*spi);

 /\*

 \* Used to enable core support for DMA handling, if can\_dma()

 \* exists and returns true then the transfer will be mapped

 \* prior to transfer\_one() being called. The driver should

 \* not modify or store xfer and dma\_tx and dma\_rx must be set

 \* while the device is prepared.

 \*/

 bool (\*can\_dma)(struct spi\_master \*master,

 struct spi\_device \*spi,

 struct spi\_transfer \*xfer);

 /\*

 \* These hooks are for drivers that want to use the generic

 \* master transfer queueing mechanism. If these are used, the

 \* transfer() function above must NOT be specified by the driver.

 \* Over time we expect SPI drivers to be phased over to this API.

 \*/

 bool queued;

 struct kthread\_worker kworker;

 struct task\_struct \*kworker\_task;

 struct kthread\_work pump\_messages;

 spinlock\_t queue\_lock;

 struct list\_head queue;

 struct spi\_message \*cur\_msg;

 bool idling;

 bool busy;

 bool running;

 bool rt;

 bool auto\_runtime\_pm;

 bool cur\_msg\_prepared;

 bool cur\_msg\_mapped;

 struct completion xfer\_completion;

 size\_t max\_dma\_len;

 int (\*prepare\_transfer\_hardware)(struct spi\_master \*master);

 **int (\*transfer\_one\_message)(struct spi\_master \*master,**

 **struct spi\_message \*mesg);**

 int (\*unprepare\_transfer\_hardware)(struct spi\_master \*master);

 int (\*prepare\_message)(struct spi\_master \*master,

 struct spi\_message \*message);

 int (\*unprepare\_message)(struct spi\_master \*master,

 struct spi\_message \*message);

 /\*

 \* These hooks are for drivers that use a generic implementation

 \* of transfer\_one\_message() provied by the core.

 \*/

 void (\*set\_cs)(struct spi\_device \*spi, bool enable);

 int (\*transfer\_one)(struct spi\_master \*master, struct spi\_device \*spi,

 struct spi\_transfer \*transfer);

 void (\*handle\_err)(struct spi\_master \*master,

 struct spi\_message \*message);

 /\* gpio chip select \*/

 int \*cs\_gpios;

 /\* DMA channels for use with core dmaengine helpers \*/

 struct dma\_chan \*dma\_tx;

 struct dma\_chan \*dma\_rx;

 /\* dummy data for full duplex devices \*/

 void \*dummy\_rx;

 void \*dummy\_tx;

};

###  2、spi\_imx\_data

struct spi\_imx\_data {

 **struct spi\_bitbang bitbang;**

 struct completion xfer\_done;

 void \_\_iomem \*base;

 struct clk \*clk\_per;

 struct clk \*clk\_ipg;

 unsigned long spi\_clk;

 unsigned int count;

 **void (\*tx)(struct spi\_imx\_data \*);**

 **void (\*rx)(struct spi\_imx\_data \*);**

 void \*rx\_buf;

 const void \*tx\_buf;

 unsigned int txfifo; /\* number of words pushed in tx FIFO \*/

 /\* DMA \*/

 unsigned int dma\_is\_inited;

 unsigned int dma\_finished;

 bool usedma;

 u32 rx\_wml;

 u32 tx\_wml;

 u32 rxt\_wml;

 struct completion dma\_rx\_completion;

 struct completion dma\_tx\_completion;

 struct dma\_slave\_config rx\_config;

 struct dma\_slave\_config tx\_config;

 const struct spi\_imx\_devtype\_data \*devtype\_data;

 int chipselect[0];

};

### 3、spi\_bitbang

struct spi\_bitbang {

 spinlock\_t lock;

 u8 busy;

 u8 use\_dma;

 u8 flags; /\* extra spi->mode support \*/

 **struct spi\_master \*master;**

 /\* setup\_transfer() changes clock and/or wordsize to match settings

 \* for this transfer; zeroes restore defaults from spi\_device.

 \*/

 **int (\*setup\_transfer)(struct spi\_device \*spi,**

 **struct spi\_transfer \*t);**

 void (\*chipselect)(struct spi\_device \*spi, int is\_on);

#define BITBANG\_CS\_ACTIVE 1 /\* normally nCS, active low \*/

#define BITBANG\_CS\_INACTIVE 0

 /\* txrx\_bufs() may handle dma mapping for transfers that don't

 \* already have one (transfer.{tx,rx}\_dma is zero), or use PIO

 \*/

 **int (\*txrx\_bufs)(struct spi\_device \*spi, struct spi\_transfer \*t);**

 /\* txrx\_word[SPI\_MODE\_\*]() just looks like a shift register \*/

 u32 (\*txrx\_word[4])(struct spi\_device \*spi,

 unsigned nsecs,

 u32 word, u8 bits);

};

 spi\_imx\_data-> bitbang-> master

 -> setup\_transfer

 -> txrx\_bufs

 spi\_master，一般需要申请spi\_alloc\_master，释放的话就是spi\_master\_put。

需要初始化spi\_master，最终使用函数spi\_register\_master注册。

 bitbang下的spi\_imx\_setupxfer函数

 spi\_imx\_setupxfer-> spi\_imx->rx = **spi\_imx\_buf\_rx\_u8;** //最终的SPI接收函数

 ->spi\_imx->tx = **spi\_imx\_buf\_tx\_u8;**  //发送函数

 ->**spi\_imx->devtype\_data->config(spi\_imx, &config);->** **mx51\_ecspi\_config //配置6ULL的SPI控制器寄存器**

bitbang下的txrx\_bufs，示例是spi\_imx\_transfer函数：

 spi\_imx\_transfer

->spi\_imx\_pio\_transfer

 ->spi\_imx\_push(spi\_imx)

 ->spi\_imx->tx(spi\_imx)

 最终调用：spi\_bitbang\_start，

 ->**master->transfer\_one\_message = spi\_bitbang\_transfer\_one;**

 ->spi\_register\_master,向系统注册spi\_master。

 **spi\_bitbang\_transfer\_on**

 **->** **bitbang->txrx\_bufs(spi, t)=** **spi\_imx\_transfer**

 **-> spi\_imx->tx(spi\_imx)= spi\_imx\_buf\_tx\_u8.**

 **所以，经过这么复杂的操作，最终目的就是，设置spi\_master的transfer\_one\_message函数为spi\_imx\_buf\_tx\_u8。**

### 4、收发函数

 spi\_imx->rx和spi\_imx->tx，以一个字节为例：

 **spi\_imx->rx = spi\_imx\_buf\_rx\_u8;**

 **spi\_imx->tx = spi\_imx\_buf\_tx\_u8;**

5、SPI接收

SPI通过中断接收，中断处理函数为spi\_imx\_isr，此函数会调用spi\_imx->rx(spi\_imx)函数来完成具体的接收过程。

## 1.4、SPI设备驱动

 SPI设备驱动就是具体的SPI芯片驱动，比如ICM20608。

 spi\_device：每个spi\_device下都有一个spi\_master。每个SPI设备，肯定挂载到了一个SPI控制器，比如ICM20608挂载到了6ULL的ECSPI3接口上。

 spi\_driver：非常重要！本讲重点，申请或者 定义一个spi\_driver，然后初始化spi\_driver中的各个成员变量，当SPI设备和驱动匹配以后，spi\_driver下的probe函数就会执行！

 spi\_driver初始化成功以后需要向内核注册，函数为：spi\_register\_driver，当注销驱动的时候需要spi\_unregister\_driver

# 二、驱动编写与测试

 1、修改设备树，添加IO相关信息

 ECSPI3\_SCLK -> UART2\_RXD

 ECSPI3\_MOSI -> UART2\_CTS

 ECSPI3\_SS0 -> UART2\_TXD

 ECSPI3\_MISO -> UART2\_RTS

 片选信号不作为硬件片选，而是作为普通的GPIO，我们在程序里面自行控制片选引脚。

 2、在ECSPI3节点下创建icm20608子节点

 3、需要初始化icm20608芯片、然后从里面读取原始数据！这个过程就要用到如何使用linux内的SPI驱动API来读写ICM20608

 用到两个重要的结构体：spi\_transfer和spi\_message

 spi\_transfer用来构建收发数据内容。

 构建spi\_transfer，然后将其打包到spi\_message里面，需要使用spi\_message\_init初始化spi\_message，然后在使用spi\_message\_add\_tail将spi\_transfer添加到spi\_message里面，最终使用spi\_sync和spi\_async来发送。