# 驱动框架

我们在i2c设备驱动中实现了tmp75\_write\_reg/tmp75\_read\_reg完成对tmp75寄存器的读写操作，里面其实就是创建i2c消息然后调用i2c\_transfer ，如果找一份3.1版本以前的内核代码可以看到里面充斥着大量的类似的代码，实现spi的设备驱动时也有同样的问题，但是其实不同的i2c设备驱动或者spi设备驱动对芯片的寄存器操作是有很多共性的。另外有的芯片同时拥有i2c与spi接口，如果硬件接口更改，相应的寄存器操作方法也需要跟着更改。针对这些问题，linux内核在3.1版本后引入regmap子系统，统一了对寄存器的操作接口，提高了代码的重用性与一致性，并且regmap在驱动和硬件寄存器之前增加了cache，减少了I/O操作次数，提高了访问效率。

regmap的框架如下：

rbtree

spi

flat

rbtree

lzo

i2c

物理总线

regmap API

remap cache

regmap\_write

regmap\_read

regmap框架分为三层：

1. api层为驱动编写人员提供了统一的接口操作寄存器。
2. cache层提供了缓存机制，减少I/O操作，当然也可以跳过cache。
3. 总线层就是对不同物理总线的操作方法的封装，目前regmap支持的物理总线包括i2c、i3c、spi、mmio、sccb、sdw、slimbus、irq、spmi、w1。

## API说明

struct regmap \* devm\_regmap\_init\_spi(struct spi\_device \*spi, const struct regmap\_config);

基于spi初始化regmap。

struct regmap \* devm\_regmap\_init\_i2c(struct i2c\_client \*i2c, const struct regmap\_config);

基于i2c初始化regmap。

void regmap\_exit(struct regmap \*map);

如果初始化时没有使用带有devm的初始化函数，则需要释放。

int regmap\_write(struct regmap \*map, unsigned int reg, unsigned int val);

写单个寄存器。

int regmap\_raw\_write(struct regmap \*map, unsigned int reg, const void \*val, size\_t val\_len);

向寄存器写入val\_len长度的数据。

int regmap\_bulk\_write(struct regmap \*map, unsigned int reg, const void \*val,size\_t val\_count);

批量写寄存器，与regmap\_raw\_write的区别是会考虑寄存器的步进信息reg\_stride。

int regmap\_update\_bits(struct regmap \*map, int reg, int mask, int val);

更新寄存器值指定bit。

int regmap\_read(struct regmap \*map, unsigned int reg, unsigned int \*val);

读单个寄存器。

int regmap\_raw\_read(struct regmap \*map, unsigned int reg, void \*val, size\_t val\_len);

从寄存器读取val\_len长度的数据。

int regmap\_bulk\_read(struct regmap \*map, int reg, void \*val, size\_t val\_count);

批量读寄存器。

## 缓存

目前regmap提供三种缓存类型分别为数组(flat)、lzo压缩以及红黑树。

* + 数组是最简单的缓存类型，当设备寄存器很少时，可以用这种类型来缓存寄存器值。
  + LZO是 Linux 中经常用到的一种压缩算法，Linux 编译后就会用这个算法来压缩。这个算法有 3 个特性：压缩快，解压不需要额外内存，压缩比可以自动调节。在这里，你可以理解为一个数组缓存，套了一层压缩，来节约内存。当设备寄存器数量中等时，可以考虑这种缓存类型。
  + 红黑树特性就是索引快，所以当设备寄存器数量比较大，或者对寄存器操作延时要求低时，就可以用这种缓存类型。

regmap的cache简单理解就是我们在寄存器写入时不仅要写到设备中，也会写入缓存，此时如果我们读取寄存器如果没有跳过缓存就可以直接从缓冲中读取，就不用读取设备了，也就减少了I/O操作。

regmap的cache操作方法通过regcache\_ops描述：

struct regcache\_ops {

const char \*name; //cache名称

enum regcache\_type type;//cache类型

int (\*init)(struct regmap \*map);//初始化

int (\*exit)(struct regmap \*map);//释放

#ifdef CONFIG\_DEBUG\_FS

void (\*debugfs\_init)(struct regmap \*map);

#endif

int (\*read)(struct regmap \*map, unsigned int reg, unsigned int \*value);//cache读

int (\*write)(struct regmap \*map, unsigned int reg, unsigned int value);//cache写

int (\*sync)(struct regmap \*map, unsigned int min, unsigned int max);//cache同步

int (\*drop)(struct regmap \*map, unsigned int min, unsigned int max);

};

flat的缓存方式很好理解，就是申请一个数组用来存寄存器值，我们在读取寄存器时直接拿到的就是数组中放好的数据，具体实现在regcache-flat.c中，我们来分析一下他的实现：

static int regcache\_flat\_init(struct regmap \*map)

{

//根据寄存器信息申请内存

map->cache = kcalloc(regcache\_flat\_get\_index(map, map->max\_register) + 1, sizeof(unsigned int), GFP\_KERNEL);

}

static int regcache\_flat\_read(struct regmap \*map,

unsigned int reg, unsigned int \*value)

{

//直接将缓存中的数返回

\*value = cache[regcache\_flat\_get\_index(map, reg)];

}

static int regcache\_flat\_write(struct regmap \*map, unsigned int reg, unsigned int value)

{

//在regmap写入时，如果未跳过cache，则会将寄存器值写入cache

cache[regcache\_flat\_get\_index(map, reg)] = value;

}

## 物理总线

regmap使用regmap\_bus结构来描述一种物理总线的读写操作方法与配置：

struct regmap\_bus {

bool fast\_io;//true则使用自旋锁，否则使用互斥锁作为锁机制

regmap\_hw\_write write;//写count个字节数据

//往指定寄存器写入数据，可以写多个寄存器+多个数据

regmap\_hw\_gather\_write gather\_write;

regmap\_hw\_async\_write async\_write;//异步写

regmap\_hw\_reg\_write reg\_write;//写单个寄存器

regmap\_hw\_reg\_update\_bits reg\_update\_bits;//更新寄存器指定bit

regmap\_hw\_read read;//读count个字节数据

regmap\_hw\_reg\_read reg\_read;//读单个寄存器

regmap\_hw\_free\_context free\_context;

regmap\_hw\_async\_alloc async\_alloc;

u8 read\_flag\_mask;//默认读寄存器掩码

enum regmap\_endian reg\_format\_endian\_default;//默认寄存器地址大小端

enum regmap\_endian val\_format\_endian\_default;//默认寄存器值大小端

size\_t max\_raw\_read;

size\_t max\_raw\_write;

};

当然关于总线的读写实现与配置无需我们关心，但是了解此结构有助于我们理解regmap读写操作的具体实现。我们以i2c为例说明regmap\_bus的实现，regmap-i2c.c中实现了包括i2c、smbus的regmap\_bus：

static struct regmap\_bus regmap\_i2c = {

.write = regmap\_i2c\_write,

.gather\_write = regmap\_i2c\_gather\_write,

.read = regmap\_i2c\_read,

.reg\_format\_endian\_default = REGMAP\_ENDIAN\_BIG,

.val\_format\_endian\_default = REGMAP\_ENDIAN\_BIG,

};

其中的读写操作分别为：

static int regmap\_i2c\_write(void \*context, const void \*data, size\_t count)

{

ret = i2c\_master\_send(i2c, data, count);

}

static int regmap\_i2c\_read(void \*context,

const void \*reg, size\_t reg\_size,

void \*val, size\_t val\_size)

{

struct i2c\_msg xfer[2];

xfer[0].addr = i2c->addr;

xfer[0].flags = 0;

xfer[0].len = reg\_size;

xfer[0].buf = (void \*)reg;

xfer[1].addr = i2c->addr;

xfer[1].flags = I2C\_M\_RD;

xfer[1].len = val\_size;

xfer[1].buf = val;

ret = i2c\_transfer(i2c->adapter, xfer, 2);

}

是不是很眼熟，这与我们在讲i2c时对tmp75的寄存器操作封装的tmp75\_write\_reg/tmp75\_read\_reg是一致的，regmap帮我们完成了这部分工作，所以我们才能只需调用regmap api就能完成寄存器读写。

实现了regmap\_bus，再加上设备寄存器的映射配置信息，就可以调用\_\_devm\_regmap\_init完成regmap的初始化。

# 寄存器映射配置

我们在进行regmap初始化时，需要传递设备的寄存器配置信息，配置信息正确才能保证我们正确读写寄存器：

struct regmap\_config {

const char \*name;//regmap名称，当一个设备有多个寄存器域时使用

int reg\_bits;//寄存器地址位数，必须配置

int reg\_stride;//寄存器地址步进，默认为1

int pad\_bits;//寄存器和值之间的填充位数

int val\_bits;//寄存器值的位数，必须设置

//判断寄存器是否可写，返回true可写，regmap子系统写入寄存器时都会调用此函数，如果writeable\_reg未定义，那么regmap子系统会检查wr\_table，判断寄存器是否在wr\_table提供的范围内，是才会执行写操作，当然wr\_table也可不定义。

bool (\*writeable\_reg)(struct device \*dev, unsigned int reg);

//判断寄存器是否可读

bool (\*readable\_reg)(struct device \*dev, unsigned int reg);

//判断寄存器是否可直接读写（cacheable），返回false,则仅使用缓冲方法，否则，寄存器直接读写。

bool (\*volatile\_reg)(struct device \*dev, unsigned int reg);

bool (\*precious\_reg)(struct device \*dev, unsigned int reg);

regmap\_lock lock;//自定义的读写前的锁，不定义则使用regmap自己的锁

regmap\_unlock unlock;//解锁

void \*lock\_arg;//锁参数

//覆盖regmap中定义的读写操作，一般用不到

int (\*reg\_read)(void \*context, unsigned int reg, unsigned int \*val);

int (\*reg\_write)(void \*context, unsigned int reg, unsigned int val);

//true则使用自旋锁，否则regmap默认使用互斥锁

bool fast\_io;

//寄存器地址最大值，读写地址小于等于此值才执行，等于0时忽略

unsigned int max\_register;

const struct regmap\_access\_table \*wr\_table;//可写寄存器表

const struct regmap\_access\_table \*rd\_table;//可读寄存器表

const struct regmap\_access\_table \*volatile\_table;

const struct regmap\_access\_table \*precious\_table;

const struct reg\_default \*reg\_defaults;

unsigned int num\_reg\_defaults;

enum regcache\_type cache\_type;//缓存类型

const void \*reg\_defaults\_raw;

unsigned int num\_reg\_defaults\_raw;

//读寄存器掩码，比如spi器件很多要通过在寄存器上或上读写标志来区分读写

unsigned long read\_flag\_mask;

//写寄存器掩码，如果读写掩码都未设置，则用regmap\_bus中的mask默认值

unsigned long write\_flag\_mask;

bool use\_single\_rw;//如果为true，批量（bulk）读写就会转换为一系列单一的读写操作，对于不支持批量读写的设备需要设置

bool can\_multi\_write;//如果为true表示支持多写模式，否则多写会分成 多次写入。

//寄存器地址大小端，大于8位时需要设置，不设置时默认使用regmap\_bus的reg\_format\_endian\_default值

enum regmap\_endian reg\_format\_endian;

//寄存器值大小端，大于8位时需要设置，不设置时默认使用regmap\_bus的val\_format\_endian\_default值

enum regmap\_endian val\_format\_endian;

const struct regmap\_range\_cfg \*ranges;

unsigned int num\_ranges;

};

regmap缓冲类型定义如下：

enum regcache\_type {

REGCACHE\_NONE,//跳过缓存

REGCACHE\_RBTREE,//红黑树

REGCACHE\_COMPRESSED,//压缩

REGCACHE\_FLAT,//数组

};

# 示例说明

示例中只是对讲解i2c时的tmp75驱动稍加修改，添加了tmp75的寄存器映射配置tmp75\_regmap\_config，然后进行了regmap初始化devm\_regmap\_init\_i2c，此后对tmp75的读写操作都替换为regmap的API。测试程序可以直接使用测试i2c设备驱动时的test\_tmp75\_driver.c，运行结果是一样的。