MPU6000读取过程

rc.sensors中

if mpu6000 -X -R 4 start

then

Firmware/src/drivers/mpu6000/ mpu6000.cpp中

extern "C" { \_\_EXPORT int mpu6000\_main(int argc, char \*argv[]); }

int mpu6000\_main(int argc, char \*argv[])

{

……

if (!strcmp(verb, "start")) {

mpu6000::start(external\_bus, rotation, accel\_range, device\_type);

}

……

}

void

start(bool external\_bus, enum Rotation rotation, int range, int device\_type)

{

//路径

int fd;

MPU6000 \*\*g\_dev\_ptr = external\_bus ? &g\_dev\_ext : &g\_dev\_int;

const char \*path\_accel = external\_bus ? MPU\_DEVICE\_PATH\_ACCEL\_EXT : MPU\_DEVICE\_PATH\_ACCEL;

const char \*path\_gyro = external\_bus ? MPU\_DEVICE\_PATH\_GYRO\_EXT : MPU\_DEVICE\_PATH\_GYRO;

if (\*g\_dev\_ptr != nullptr)

/\* if already started, the still command succeeded \*/

{

errx(0, "already started");

}

/\* create the driver \*/

if (external\_bus) {

#ifdef PX4\_SPI\_BUS\_EXT

# if defined(PX4\_SPIDEV\_EXT\_ICM)

spi\_dev\_e cs = (spi\_dev\_e)(device\_type == 6000 ? PX4\_SPIDEV\_EXT\_MPU : PX4\_SPIDEV\_EXT\_ICM);

# else

spi\_dev\_e cs = (spi\_dev\_e) PX4\_SPIDEV\_EXT\_MPU;

# endif

\*g\_dev\_ptr = new MPU6000(PX4\_SPI\_BUS\_EXT, path\_accel, path\_gyro, cs, rotation, device\_type);

#else

errx(0, "External SPI not available");

#endif

} else {

#if defined(PX4\_SPIDEV\_ICM)

spi\_dev\_e cs = (spi\_dev\_e)(device\_type == 6000 ? PX4\_SPIDEV\_MPU : PX4\_SPIDEV\_ICM);

#else

spi\_dev\_e cs = (spi\_dev\_e) PX4\_SPIDEV\_MPU;

#endif

\*g\_dev\_ptr = new MPU6000(PX4\_SPI\_BUS\_SENSORS, path\_accel, path\_gyro, cs, rotation, device\_type);

}

if (\*g\_dev\_ptr == nullptr) {

goto fail;

}

//注册设备名

if (OK != (\*g\_dev\_ptr)->init()) {

goto fail;

}

//打开设备

/\* set the poll rate to default, starts automatic data collection \*/

fd = open(path\_accel, O\_RDONLY);

if (fd < 0) {

goto fail;

}

//这里操作传感器

if (ioctl(fd, SENSORIOCSPOLLRATE, SENSOR\_POLLRATE\_DEFAULT) < 0) {

goto fail;

}

if (ioctl(fd, ACCELIOCSRANGE, range) < 0) {

goto fail;

}

close(fd);

exit(0);

fail:

if (\*g\_dev\_ptr != nullptr) {

delete(\*g\_dev\_ptr);

\*g\_dev\_ptr = nullptr;

}

errx(1, "no device on this bus");

}

跳到init()中

int

MPU6000::init()

{

int ret;

/\* do SPI init (and probe) first \*/

ret = SPI::init();

/\* if probe/setup failed, bail now \*/

if (ret != OK) {

DEVICE\_DEBUG("SPI setup failed");

return ret;

}

/\* allocate basic report buffers \*/

\_accel\_reports = new ringbuffer::RingBuffer(2, sizeof(accel\_report));

if (\_accel\_reports == nullptr) {

goto out;

}

\_gyro\_reports = new ringbuffer::RingBuffer(2, sizeof(gyro\_report));

if (\_gyro\_reports == nullptr) {

goto out;

}

if (reset() != OK) {

goto out;

}

/\* Initialize offsets and scales \*/

\_accel\_scale.x\_offset = 0;

\_accel\_scale.x\_scale = 1.0f;

\_accel\_scale.y\_offset = 0;

\_accel\_scale.y\_scale = 1.0f;

\_accel\_scale.z\_offset = 0;

\_accel\_scale.z\_scale = 1.0f;

\_gyro\_scale.x\_offset = 0;

\_gyro\_scale.x\_scale = 1.0f;

\_gyro\_scale.y\_offset = 0;

\_gyro\_scale.y\_scale = 1.0f;

\_gyro\_scale.z\_offset = 0;

\_gyro\_scale.z\_scale = 1.0f;

/\* do CDev init for the gyro device node, keep it optional \*/

ret = \_gyro->init();

/\* if probe/setup failed, bail now \*/

if (ret != OK) {

DEVICE\_DEBUG("gyro init failed");

return ret;

}

\_accel\_class\_instance = register\_class\_devname(ACCEL\_BASE\_DEVICE\_PATH);

measure();

/\* advertise sensor topic, measure manually to initialize valid report \*/

struct accel\_report arp;

\_accel\_reports->get(&arp);

/\* measurement will have generated a report, publish \*/

\_accel\_topic = orb\_advertise\_multi(ORB\_ID(sensor\_accel), &arp,

&\_accel\_orb\_class\_instance, (is\_external()) ? ORB\_PRIO\_MAX : ORB\_PRIO\_HIGH);

if (\_accel\_topic == nullptr) {

warnx("ADVERT FAIL");

}

/\* advertise sensor topic, measure manually to initialize valid report \*/

struct gyro\_report grp;

\_gyro\_reports->get(&grp);

\_gyro->\_gyro\_topic = orb\_advertise\_multi(ORB\_ID(sensor\_gyro), &grp,

&\_gyro->\_gyro\_orb\_class\_instance, (is\_external()) ? ORB\_PRIO\_MAX : ORB\_PRIO\_HIGH);

if (\_gyro->\_gyro\_topic == nullptr) {

warnx("ADVERT FAIL");

}

out:

return ret;

}

与init()并列的ioctl()中

int

MPU6000::ioctl(struct file \*filp, int cmd, unsigned long arg)

{

switch (cmd) {

case SENSORIOCRESET:

return reset();

case SENSORIOCSPOLLRATE: {

switch (arg) {

/\* switching to manual polling \*/

case SENSOR\_POLLRATE\_MANUAL:

stop();

\_call\_interval = 0;

return OK;

/\* external signalling not supported \*/

case SENSOR\_POLLRATE\_EXTERNAL:

/\* zero would be bad \*/

case 0:

return -EINVAL;

/\* set default/max polling rate \*/

case SENSOR\_POLLRATE\_MAX:

return ioctl(filp, SENSORIOCSPOLLRATE, 1000);

case SENSOR\_POLLRATE\_DEFAULT:

return ioctl(filp, SENSORIOCSPOLLRATE, MPU6000\_ACCEL\_DEFAULT\_RATE);

/\* adjust to a legal polling interval in Hz \*/

/\* 设置measure的采样周期 \*/

default: {

/\* do we need to start internal polling? \*/

bool want\_start = (\_call\_interval == 0);

/\* convert hz to hrt interval via microseconds \*/

unsigned ticks = 1000000 / arg;

/\* check against maximum sane rate \*/

if (ticks < 1000) {

return -EINVAL;

}

// adjust filters

float cutoff\_freq\_hz = \_accel\_filter\_x.get\_cutoff\_freq();

float sample\_rate = 1.0e6f / ticks;

\_set\_dlpf\_filter(cutoff\_freq\_hz);

\_accel\_filter\_x.set\_cutoff\_frequency(sample\_rate, cutoff\_freq\_hz);

\_accel\_filter\_y.set\_cutoff\_frequency(sample\_rate, cutoff\_freq\_hz);

\_accel\_filter\_z.set\_cutoff\_frequency(sample\_rate, cutoff\_freq\_hz);

float cutoff\_freq\_hz\_gyro = \_gyro\_filter\_x.get\_cutoff\_freq();

\_set\_dlpf\_filter(cutoff\_freq\_hz\_gyro);

\_gyro\_filter\_x.set\_cutoff\_frequency(sample\_rate, cutoff\_freq\_hz\_gyro);

\_gyro\_filter\_y.set\_cutoff\_frequency(sample\_rate, cutoff\_freq\_hz\_gyro);

\_gyro\_filter\_z.set\_cutoff\_frequency(sample\_rate, cutoff\_freq\_hz\_gyro);

/\* update interval for next measurement \*/

/\* XXX this is a bit shady, but no other way to adjust... \*/

\_call\_interval = ticks;

/\*

set call interval faster then the sample time. We

then detect when we have duplicate samples and reject

them. This prevents aliasing due to a beat between the

stm32 clock and the mpu6000 clock

\*/

\_call.period = \_call\_interval - MPU6000\_TIMER\_REDUCTION;

/\* if we need to start the poll state machine, do it \*/

/\* 这里进行measure \*/

if (want\_start) {

start();

}

return OK;

}

}

}

case SENSORIOCGPOLLRATE:

if (\_call\_interval == 0) {

return SENSOR\_POLLRATE\_MANUAL;

}

return 1000000 / \_call\_interval;

case SENSORIOCSQUEUEDEPTH: {

/\* lower bound is mandatory, upper bound is a sanity check \*/

if ((arg < 1) || (arg > 100)) {

return -EINVAL;

}

irqstate\_t flags = px4\_enter\_critical\_section();

if (!\_accel\_reports->resize(arg)) {

px4\_leave\_critical\_section(flags);

return -ENOMEM;

}

px4\_leave\_critical\_section(flags);

return OK;

}

case SENSORIOCGQUEUEDEPTH:

return \_accel\_reports->size();

case ACCELIOCGSAMPLERATE:

return \_sample\_rate;

case ACCELIOCSSAMPLERATE:

\_set\_sample\_rate(arg);

return OK;

case ACCELIOCGLOWPASS:

return \_accel\_filter\_x.get\_cutoff\_freq();

case ACCELIOCSLOWPASS:

// set hardware filtering

\_set\_dlpf\_filter(arg);

// set software filtering

\_accel\_filter\_x.set\_cutoff\_frequency(1.0e6f / \_call\_interval, arg);

\_accel\_filter\_y.set\_cutoff\_frequency(1.0e6f / \_call\_interval, arg);

\_accel\_filter\_z.set\_cutoff\_frequency(1.0e6f / \_call\_interval, arg);

return OK;

case ACCELIOCSSCALE: {

/\* copy scale, but only if off by a few percent \*/

struct accel\_calibration\_s \*s = (struct accel\_calibration\_s \*) arg;

float sum = s->x\_scale + s->y\_scale + s->z\_scale;

if (sum > 2.0f && sum < 4.0f) {

memcpy(&\_accel\_scale, s, sizeof(\_accel\_scale));

return OK;

} else {

return -EINVAL;

}

}

case ACCELIOCGSCALE:

/\* copy scale out \*/

memcpy((struct accel\_calibration\_s \*) arg, &\_accel\_scale, sizeof(\_accel\_scale));

return OK;

case ACCELIOCSRANGE:

return set\_accel\_range(arg);

case ACCELIOCGRANGE:

return (unsigned long)((\_accel\_range\_m\_s2) / MPU6000\_ONE\_G + 0.5f);

case ACCELIOCSELFTEST:

return accel\_self\_test();

default:

/\* give it to the superclass \*/

return SPI::ioctl(filp, cmd, arg);

}

}

进入start()

void

MPU6000::start()

{

/\* make sure we are stopped first \*/

stop();

/\* discard any stale data in the buffers \*/

\_accel\_reports->flush();

\_gyro\_reports->flush();

/\* start polling at the specified rate \*/

hrt\_call\_every(&\_call,

1000,

\_call\_interval - MPU6000\_TIMER\_REDUCTION,

(hrt\_callout)&MPU6000::measure\_trampoline, this);

}

进入measure\_trampoline()

void

MPU6000::measure\_trampoline(void \*arg)

{

MPU6000 \*dev = reinterpret\_cast<MPU6000 \*>(arg);

/\* make another measurement \*/

dev->measure();

}

该函数实现周期延时100ms后以\_call\_interval 这个周期调用MPU6000::measure\_trampoline这个函数。而measure\_trampoline又调用measure。从而实现一直读取传感器的数值。即我们可以通过调用**MPU6000::ioctl**来调用**measure**函数。

进入measure()

IMU380::measure()

{

...

//读取传感器的值

/\*

\* Fetch the full set of measurements from the MPU6000 in one pass.

\*/

mpu\_report.cmd = DIR\_READ | MPUREG\_INT\_STATUS;

// sensor transfer at high clock speed

//时钟频率

set\_frequency(MPU6000\_HIGH\_BUS\_SPEED);

//读传感器

if (OK != transfer((uint8\_t \*)&mpu\_report, ((uint8\_t \*)&mpu\_report), sizeof(mpu\_report))) {

return;

}

check\_registers(); ..

//将加速度计的值转化为加速度单位 m/s^2.

\_accel\_range\_scale = 1.0f / 4000.0f;

float x\_in\_new = ((xraw\_f \* \_accel\_range\_scale) - \_accel\_scale.x\_offset) \* \_accel\_scale.x\_scale;

float y\_in\_new = ((yraw\_f \* \_accel\_range\_scale) - \_accel\_scale.y\_offset) \* \_accel\_scale.y\_scale;

float z\_in\_new = ((zraw\_f \* \_accel\_range\_scale) - \_accel\_scale.z\_offset) \* \_accel\_scale.z\_scale;

...

//滤波

arb.x = \_accel\_filter\_x.apply(x\_in\_new);

arb.y = \_accel\_filter\_y.apply(y\_in\_new);

arb.z = \_accel\_filter\_z.apply(z\_in\_new);

//积分

math::Vector<3> aval(x\_in\_new, y\_in\_new, z\_in\_new);

math::Vector<3> aval\_integrated;

bool accel\_notify = \_accel\_int.put(arb.timestamp, aval, aval\_integrated, arb.integral\_dt);

arb.x\_integral = aval\_integrated(0);

arb.y\_integral = aval\_integrated(1);

arb.z\_integral = aval\_integrated(2);

...

//此处陀螺仪类似

...

//最后通过消息发送出去

if (accel\_notify && !(\_pub\_blocked)) {

/\* publish it \*/

orb\_publish(ORB\_ID(sensor\_accel), \_accel\_topic, &arb);

}

if (gyro\_notify && !(\_pub\_blocked)) {

/\* publish it \*/

orb\_publish(ORB\_ID(sensor\_gyro), \_gyro->\_gyro\_topic, &grb);

}

}

有以下几点得知道在哪设置：

读传感器的时钟频率:set\_frequency(MPU6000\_HIGH\_BUS\_SPEED);

传感器采样周期：ioctl()中的default里面的\_call\_interval值

用逻辑分析仪分析

SPI驱动结构

接着MPU6000读取过程

进入if (OK != transfer((uint8\_t \*)&mpu\_report, ((uint8\_t \*)&mpu\_report), sizeof(mpu\_report)))这里的transfer()

int

SPI::transfer(uint8\_t \*send, uint8\_t \*recv, unsigned len)

{

int result;

if ((send == nullptr) && (recv == nullptr)) {

return -EINVAL;

}

LockMode mode = up\_interrupt\_context() ? LOCK\_NONE : locking\_mode;

/\* lock the bus as required \*/

switch (mode) {

default:

case LOCK\_PREEMPTION: {

irqstate\_t state = px4\_enter\_critical\_section();

result = \_transfer(send, recv, len);//读取

px4\_leave\_critical\_section(state);

}

break;

case LOCK\_THREADS:

SPI\_LOCK(\_dev, true);

result = \_transfer(send, recv, len); //读取

SPI\_LOCK(\_dev, false);

break;

case LOCK\_NONE:

result = \_transfer(send, recv, len); //读取

break;

}

return result;

}

进入\_transfer(send, recv, len)

int

SPI::\_transfer(uint8\_t \*send, uint8\_t \*recv, unsigned len)

{

SPI\_SETFREQUENCY(\_dev, \_frequency);

SPI\_SETMODE(\_dev, \_mode);

SPI\_SETBITS(\_dev, 8);

SPI\_SELECT(\_dev, \_device, true);

/\* do the transfer \*/

SPI\_EXCHANGE(\_dev, send, recv, len);

/\* and clean up \*/

SPI\_SELECT(\_dev, \_device, false);

return OK;

}

这个里面的函数随便挑一个进入都会跟到Fiemware/build\_px4fmu-v2\_default/px4fmu-v2/Nuttx/nuttx/include/nuttx/spi.h里面来

比如进入SPI\_SETFREQUENCY(\_dev, \_frequency);

跟到了#define SPI\_SETFREQUENCY(d,f) ((d)->ops->setfrequency(d,f))

再进入((d)->ops->setfrequency(d,f))

跟到了uint32\_t (\*setfrequency)(FAR struct spi\_dev\_s \*dev, uint32\_t frequency);

struct spi\_ops\_s

{

#ifndef CONFIG\_SPI\_OWNBUS

int (\*lock)(FAR struct spi\_dev\_s \*dev, bool lock);

#endif

void (\*select)(FAR struct spi\_dev\_s \*dev, enum spi\_dev\_e devid,

bool selected);

uint32\_t (\*setfrequency)(FAR struct spi\_dev\_s \*dev, uint32\_t frequency);

void (\*setmode)(FAR struct spi\_dev\_s \*dev, enum spi\_mode\_e mode);

void (\*setbits)(FAR struct spi\_dev\_s \*dev, int nbits);

uint8\_t (\*status)(FAR struct spi\_dev\_s \*dev, enum spi\_dev\_e devid);

#ifdef CONFIG\_SPI\_CMDDATA

int (\*cmddata)(FAR struct spi\_dev\_s \*dev, enum spi\_dev\_e devid, bool cmd);

#endif

uint16\_t (\*send)(FAR struct spi\_dev\_s \*dev, uint16\_t wd);

#ifdef CONFIG\_SPI\_EXCHANGE

void (\*exchange)(FAR struct spi\_dev\_s \*dev, FAR const void \*txbuffer,

FAR void \*rxbuffer, size\_t nwords);

#else

void (\*sndblock)(FAR struct spi\_dev\_s \*dev, FAR const void \*buffer,

size\_t nwords);

void (\*recvblock)(FAR struct spi\_dev\_s \*dev, FAR void \*buffer,

size\_t nwords);

#endif

int (\*registercallback)(FAR struct spi\_dev\_s \*dev, spi\_mediachange\_t callback,

void \*arg);

};

struct spi\_dev\_s

{

const struct spi\_ops\_s \*ops;

};

build\_px4fmu-v2\_default这个目录下都是编译时自动拷贝系统文件生成的，其实build\_px4fmu-v2\_default/px4fmu-v2/Nuttx/nuttx/include/nuttx/spi.h来自于Fiemware/Nuttx/nuttx/include/nuttx/spi.h

这是nuttx中关于spi的抽象，定义了spi操作的所有结构体和API。是对spi对象最关键的抽象结构体。

可以看出，spi\_ops\_s是一个函数的结构体，**当调用这些函数时将会调用最底层stm32\_spi.c中定义的函数来完成操作。如调用exchange函数将调用stm32\_spi.c中的spi\_exchange函数。**

当把spi\_ops\_s结构体设置完后，就可以调用**最底层stm32\_spi.c，设置stm32f4的寄存器了**

**由此，关于SPI的读取数据的整个框架就非常清晰了！**

**SPI有pix的抽象层，nuttx的抽象层指针结构体，stm32的驱动层**

**接下来是初始化的框架**

px4fmu-v2中用到了三条的SPI，一条SPI1，一条SPI2和一条SPI4，其中SPI1用作传感器，SPI2用作RAM，SPI4作为外部引出。具体定义在Firmwar/src/driver/boards/px4fmu-v2/board\_config.h中有

#define PX4\_SPI\_BUS\_SENSORS 1

#define PX4\_SPI\_BUS\_RAMTRON 2

#define PX4\_SPI\_BUS\_EXT 4

#define PX4\_SPI\_BUS\_BARO PX4\_SPI\_BUS\_SENSORS

在Firmware/src/drivers/mpu6000/mpu6000.cpp中

void

start(bool external\_bus, enum Rotation rotation, int range, int device\_type)

{

……

}

可以看到mpu6000用的是外部SPI

start(bool external\_bus, enum Rotation rotation, int range, int device\_type)

{

……

\*g\_dev\_ptr = new MPU6000(PX4\_SPI\_BUS\_SENSORS, path\_accel, path\_gyro, cs, rotation, device\_type);

……

if (OK != (\*g\_dev\_ptr)->init())

……

}

start函数就是调用new MPU6000(PX4\_SPI\_BUS\_SENSORS, path\_accel, path\_gyro, cs, rotation, device\_type);的构造函数，而我们在构造函数中发现有

MPU6000::MPU6000(int bus, const char \*path\_accel, const char \*path\_gyro, spi\_dev\_e device, enum Rotation rotation, int device\_type) :

SPI("MPU6000", path\_accel, bus, device, SPIDEV\_MODE3, MPU6000\_LOW\_BUS\_SPEED),

……

也就是调用MPU6000构造函数的时候也会调用SPI的构造函数。

接着进入(\*g\_dev\_ptr)->init()

Int MPU6000::init()

{

int ret;

/\* do SPI init (and probe) first \*/

ret = SPI::init();

……

}

再进入SPI::init()

int SPI::init()

{

......

if (\_dev == nullptr) {

\_dev = px4\_spibus\_initialize(\_bus);

}

......

/\* do base class init, which will create the device node, etc. \*/

ret = CDev::init();

......

}

该函数\_dev = px4\_spibus\_initialize(\_bus);在**src/platforms/px4\_micro\_hal.h 中定义的**

#  define px4\_spibus\_initialize(port\_1based)       up\_spiinitialize(port\_1based)

up\_spiinitialize()在Firmware/Nuttx/nuttx/include/nuttx/spi.h中声明，函数实现在Firmware/Nuttx/nuttx/arch/arm/src/stm32/stm32\_spi.c

FAR struct spi\_dev\_s \*up\_spiinitialize(int port)

{

FAR struct stm32\_spidev\_s \*priv = NULL;

irqstate\_t flags = irqsave();

#ifdef CONFIG\_STM32\_SPI1

if (port == 1)

{

/\* Select SPI1 \*/

priv = &g\_spi1dev;

/\* Only configure if the port is not already configured \*/

if ((spi\_getreg(priv, STM32\_SPI\_CR1\_OFFSET) & SPI\_CR1\_SPE) == 0)

{

/\* Configure SPI1 pins: SCK, MISO, and MOSI \*/

stm32\_configgpio(GPIO\_SPI1\_SCK);

stm32\_configgpio(GPIO\_SPI1\_MISO);

stm32\_configgpio(GPIO\_SPI1\_MOSI);

/\* Set up default configuration: Master, 8-bit, etc. \*/

spi\_portinitialize(priv);

}

}

else

#endif

#ifdef CONFIG\_STM32\_SPI2

if (port == 2)

{

/\* Select SPI2 \*/

priv = &g\_spi2dev;

/\* Only configure if the port is not already configured \*/

if ((spi\_getreg(priv, STM32\_SPI\_CR1\_OFFSET) & SPI\_CR1\_SPE) == 0)

{

/\* Configure SPI2 pins: SCK, MISO, and MOSI \*/

stm32\_configgpio(GPIO\_SPI2\_SCK);

stm32\_configgpio(GPIO\_SPI2\_MISO);

stm32\_configgpio(GPIO\_SPI2\_MOSI);

/\* Set up default configuration: Master, 8-bit, etc. \*/

spi\_portinitialize(priv);

}

}

else

#endif

#ifdef CONFIG\_STM32\_SPI3

if (port == 3)

{

/\* Select SPI3 \*/

priv = &g\_spi3dev;

/\* Only configure if the port is not already configured \*/

if ((spi\_getreg(priv, STM32\_SPI\_CR1\_OFFSET) & SPI\_CR1\_SPE) == 0)

{

/\* Configure SPI3 pins: SCK, MISO, and MOSI \*/

stm32\_configgpio(GPIO\_SPI3\_SCK);

stm32\_configgpio(GPIO\_SPI3\_MISO);

stm32\_configgpio(GPIO\_SPI3\_MOSI);

/\* Set up default configuration: Master, 8-bit, etc. \*/

spi\_portinitialize(priv);

}

}

else

#endif

#ifdef CONFIG\_STM32\_SPI4

if (port == 4)

{

/\* Select SPI4 \*/

priv = &g\_spi4dev;

/\* Only configure if the port is not already configured \*/

if ((spi\_getreg(priv, STM32\_SPI\_CR1\_OFFSET) & SPI\_CR1\_SPE) == 0)

{

/\* Configure SPI4 pins: SCK, MISO, and MOSI \*/

stm32\_configgpio(GPIO\_SPI4\_SCK);

stm32\_configgpio(GPIO\_SPI4\_MISO);

stm32\_configgpio(GPIO\_SPI4\_MOSI);

/\* Set up default configuration: Master, 8-bit, etc. \*/

spi\_portinitialize(priv);

}

}

else

#endif

#ifdef CONFIG\_STM32\_SPI5

if (port == 5)

{

/\* Select SPI5 \*/

priv = &g\_spi5dev;

/\* Only configure if the port is not already configured \*/

if ((spi\_getreg(priv, STM32\_SPI\_CR1\_OFFSET) & SPI\_CR1\_SPE) == 0)

{

/\* Configure SPI5 pins: SCK, MISO, and MOSI \*/

stm32\_configgpio(GPIO\_SPI5\_SCK);

stm32\_configgpio(GPIO\_SPI5\_MISO);

stm32\_configgpio(GPIO\_SPI5\_MOSI);

/\* Set up default configuration: Master, 8-bit, etc. \*/

spi\_portinitialize(priv);

}

}

else

#endif

#ifdef CONFIG\_STM32\_SPI6

if (port == 6)

{

/\* Select SPI6 \*/

priv = &g\_spi6dev;

/\* Only configure if the port is not already configured \*/

if ((spi\_getreg(priv, STM32\_SPI\_CR1\_OFFSET) & SPI\_CR1\_SPE) == 0)

{

/\* Configure SPI6 pins: SCK, MISO, and MOSI \*/

stm32\_configgpio(GPIO\_SPI6\_SCK);

stm32\_configgpio(GPIO\_SPI6\_MISO);

stm32\_configgpio(GPIO\_SPI6\_MOSI);

/\* Set up default configuration: Master, 8-bit, etc. \*/

spi\_portinitialize(priv);

}

}

else

#endif

{

spidbg("ERROR: Unsupported SPI port: %d\n", port);

return NULL;

}

irqrestore(flags);

return (FAR struct spi\_dev\_s \*)priv;

}

# 其次，对于需要注册到cdev中的spi来说：